A Hardware-Software Codesign Framework for Cellular Computing

THÈSE N^{O} 4354 (2009)

PRÉSENTÉE LE 5 JUIN 2009

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

GROUPE IJSPEERT

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Pierre-André MUDRY

acceptée sur proposition du jury:

Prof. E. Sanchez, président du jury
Prof. A. Ijspeert, Prof. G. Tempesti, directeurs de thèse
Dr S. Bocchio, rapporteure
Prof. G. de Micheli, rapporteur
Dr G. Sassatelli, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2009

A ceux qui sont partis
et à ceux qui restent.

Résumé

LA COURSE à la performance commencée il y a plus de trente ans dans l'industrie des processeurs voit se profiler à l'horizon d'un futur désormais proche des limites physiques qui seront difficiles à franchir. Pour maintenir le rythme de croissance de la puissance de calcul des processeurs, l'utilisation de plusieurs unités de calcul en parallèle est désormais courante. Toutefois, cette augmentation du parallélisme n'est pas sans poser son lot de problèmes liés à la concurrence ou encore au partage des ressources. De plus, peu de programmeurs possèdent aujourd'hui une connaissance suffisante de la problématique de la programmation parallèle pour faire face à l'essor de tels systèmes.

Toutefois, le recours au parallélisme dans cette quête de performance n'exclut pas d'autres pistes de recherche telles que de nouvelles méthodes de fabrication, issues par exemple des nanotechnologies. Celles-ci imposeront probablement le renouvellement des méthodologies de conception des processeurs afin de pouvoir faire face à des contraintes telles qu'un nombre accru d'erreurs matérielles ou encore la programmation de systèmes contenant plusieurs centaines de processeurs.

Une source d'inspiration possible pour répondre à de telles problématiques se trouve dans la biologie. En effet, les êtres vivants possèdent des caractéristiques intéressantes telles que la résistance aux dommages, certains organismes tels que les salamandres pouvant se régénérer en partie, ou encore une organisation dynamique, des cellules étant perpétuellement remplacées par de nouvelles. Dans cette optique, un certain nombre de travaux ont mis en évidence l'intérêt d'imiter partiellement certaines caractéristiques de ces organismes afin de les utiliser dans du matériel informatique. Ainsi, il a été possible de réaliser des systèmes relativement simples inspirés de ces mécanismes, comme une horloge auto-réparante. Toutefois, l'application de ces méthodologies pour des problèmes plus complexes reste très délicate en raison de la difficulté à utiliser la polyvalence proposée par ces systèmes matériels, inhérente à une implémentation complètement matérielle qui, par nature, est complexe à programmer.

Dans le cadre de cette thèse, nous nous proposons d'étudier cette problématique en plaçant tout d'abord au cœur de notre approche un processeur totalement flexible, possédant d'une part des caractéristiques permettant d'appliquer des mécanismes issus de la bio-inspiration et, d'autre part, étant à même d'assumer les tâches de calcul que l'on attend généralement d'un processeur. Nous verrons ainsi qu'il est possible d'obtenir un tel processeur et que l'on peut, par exemple, le faire évoluer afin de le spécialiser pour différentes applications.

Dans un deuxième temps, nous nous proposons d'analyser comment la mise en parallèle d'un grand nombre de ces processeurs sur une plateforme matérielle idoine permet d'explorer différents aspects de la bio-inspiration dans des situations réelles. Nous mettrons ainsi en lumière que le principe des architectures cellulaires permet, à l'aide de différentes abstractions logicielles, d'utiliser la puissance des processeurs tout en restant flexible. Par le biais d'une interface graphique facile d'emploi, nous illustrerons comment il est possible de simplifier la programmation de tels systèmes grâce à différents outils logiciels. Finalement, nous appliquerons notre jeu d'outils logiciels et matériels afin de montrer, sur des exemples concrets typiques des applications embarquées, comment l'auto-organisation et la réplication peuvent apporter certaines réponses au problème de la performance.

Mots-clés: *processeur TTA, Move, matériel électronique bio-inspiré, FPGA, algorithmes d'évolution, réplication, MPSoC, routage, SSSP.*

Abstract

UNTIL RECENTLY, the ever-increasing demand of computing power has been met on one hand by increasing the operating frequency of processors and on the other hand by designing architectures capable of exploiting parallelism at the instruction level through hardware mechanisms such as superscalar execution. However, both these approaches seem to have reached a plateau, mainly due to issues related to design complexity and cost-effectiveness.

To face the stabilization of performance of single-threaded processors, the current trend in processor design seems to favor a switch to coarser-grain parallelization, typically at the thread level. In other words, high computational power is achieved not only by a single, very fast and very complex processor, but through the parallel operation of several processors, each executing a different thread.

Extrapolating this trend to take into account the vast amount of on-chip hardware resources that will be available in the next few decades (either through further shrinkage of silicon fabrication processes or by the introduction of molecular-scale devices), together with the predicted features of such devices (e.g., the impossibility of global synchronization or higher failure rates), it seems reasonable to foretell that current design techniques will not be able to cope with the requirements of next-generation electronic devices and that novel design tools and programming methods will have to be devised.

A tempting source of inspiration to solve the problems implied by a massively parallel organization and inherently error-prone substrates is biology. In fact, living beings possess characteristics, such as robustness to damage and self-organization, which were shown in previous research as interesting to be implemented in hardware. For instance, it was possible to realize relatively simple systems, such as a self-repairing watch.

Overall, these bio-inspired approaches seem very promising but their interest for a wider audience is problematic because their heavily hardware-oriented designs lack some of the flexibility achievable with a general purpose processor.

In the context of this thesis, we will introduce a processor-grade processing element at the heart of a bio-inspired hardware system. This processor, based on a single-instruction, features some key properties that allow it to maintain the versatility required by the implementation of bio-inspired mechanisms and to realize general computation. We will also demonstrate that the flexibility of such a processor enables it to be evolved so it can be tailored to different types of applications.

In the second half of this thesis, we will analyze how the implementation of a large number of these processors can be used on a hardware platform to explore various bio-inspired mechanisms. Based on an extensible platform of many FPGAs, configured as a networked structure of processors, the hardware part of this computing framework is backed by an open library of software components that provides primitives for efficient inter-processor communication and distributed computation. We will show that this dual software–hardware approach allows a very quick exploration of different ways to solve computational problems using bio-inspired techniques. In addition, we also show that the flexibility of our approach allows it to exploit replication as a solution to issues that concern standard embedded applications.

Key words: *TTA processor, Move, bio-inspired hardware, FPGA, evolutionary algorithms, MPSoC, routing, replication, SSSP.*

Remerciements

RÉALISER UNE thèse est un exercice de longue haleine dont il est difficile de comprendre les tenants et les aboutissants sans l'avoir vécu personnellement. Qui plus est, la thèse est un exercice qui s'effectue majoritairement en solitaire. Toutefois, même si l'activité de la thèse elle-même ne saurait se partager, l'environnement dans lequel celle-ci est réalisée joue un rôle primordial et qui constitue la raison d'être profonde de cette page de remerciements. En effet, à quoi donc auraient ressemblé ces quatre ans sans votre support à vous, amis, connaissances et collègues qui avez rendu possible ce travail à divers niveaux ? À cette question, je ne peux répondre que par le témoignage de toute ma gratitude.

Du point de vue académique tout d'abord, celle-ci va aux membres de mon jury de thèse, à savoir Giovanni de Micheli, Gilles Sassatelli ainsi que Sara Bocchio, pour avoir amené leur expertise et leurs commentaires qui ont permis d'améliorer la version finale de ce document ainsi qu'au président du jury, Eduardo Sanchez. Merci à Gianluca Tempesti de m'avoir trouvé ce poste en me laissant une liberté totale et au Fonds national suisse de la recherche scientifique de l'avoir financé. Merci également à Auke Ijspeert de m'avoir accueilli dans son laboratoire en cours de route tout en étant une source intarissable et efficace de bons conseils. Finalement, je tiens également à remercier Daniel Mange pour sa sympathie et pour avoir pris du temps pour que ma thèse se termine en douceur.

Au niveau de mes collègues de travail, un énorme merci va à notre "bureau du bonheur", lieu mystique de découverte musicale et d'expérience de vie à part entière. Plus précisément, merci à Sarah qui a eu la gentillesse, peut-être bien malgré elle, d'écouter mes états d'âme, doutes et autres coups de gueule pendant une majeure partie de ma thèse. Nos multiples cafés ont constitué pour moi un véritable bol d'air frais et je t'en remercie. Merci également à Ludovic, pour nos festives escapades et pour avoir partagé avec moi ses expériences de fin de thèse. Je dois également tirer ma casquette à Alessandro, capable de résoudre absolument tous les problèmes informatiques, souvent même avant qu'ils apparaissent. Merci pour ton aide et ta gentillesse !

J'en profite également pour remercier les étudiants qui m'ont amené par leur travail de l'aide sur différents points de cette recherche. Plus particulièrement, un grand merci à Guillaume pour le chapitre sur l'évolution ainsi que pour le temps que nous avons passé ensemble à travailler et que j'ai beaucoup apprécié. Merci également à Julien pour son travail acharné sur CAFCA et SSSP et à Michel pour le compilateur.

Bien entendu, avant d'arriver à la thèse, mon parcours a inclus plusieurs années d'étude que j'ai eu le plaisir de traverser avec des gens qui méritent la plus grande estime. Un énorme merci donc à David, programmeur *emeritus* aux multiples traits de génie dont la dextérité au clavier n'a d'égale que la rapidité de programmation et avec qui j'ai eu l'immense honneur de travailler depuis nos laborieux débuts au Poly. Je n'ai jamais autant appris qu'avec toi et ton niveau surhumain m'a forcé à me dépasser constamment, ne serait-ce que pour pouvoir te suivre, enfin sauf en ce qui concerne les flancs d'horloge bien sûr ! Merci aussi à Leto, dont le travail a toujours réussi à conserver cette aura de magie qui fait rêver. Finalement, merci à Bertrand pour nos toujours sympathiques repas et discussions qui couvraient tant les jeux vidéo que les mathématiques les plus complexes auxquelles je faisais péniblement semblant de comprendre quelque chose.

Le travail étant une chose, il y a une vie en dehors de celui-ci et je n'en serais pas là aujourd'hui sans des gens très importants pour moi. Merci donc à Igor, pilote chevronné de presque tous les moyens de transport existants, blagueur hors pair qui a même réussi, de manière étonnante et ceci à plusieurs reprises, à me prendre dans son avion. Faire l'école buissonière et manger de la pizza avec

toi comptent clairement parmis mes activités préférées ! Merci aussi, sans aucun ordre particulier, à Antonin pour toutes nos activités campagnardes, à Maître Charvet pour ses discussions avec Rachel, à Torrent pour sa vision du monde, à l'officieux *Club de Chibre* et à ses membres pour avoir occupé bien des dimanches après-midi, à A.S. pour m'avoir montré quelques arbres dans la forêt et à *Depeche Mode* pour leur musique.

Merci encore à Jacqueline, Pascal et Laurence pour leur accueil ainsi que pour leur gentillesse depuis plus de dix ans maintenant.

Un grand merci à ma famille, dont le dévouement m'a offert l'opportunité de faire mes études et sans qui rien n'aurait été possible.

Le dernier mot sera finalement pour Rachel, incroyable amie qui m'a accompagné pendant une belle partie de mon chemin et sans qui la vie serait définitivement bien moins drôle. Merci de tout cœur, je n'y serais pas arrivé sans toi !

Contents

Chapter 1

Introduction

"It all happened much faster than we expected."

JOHN A. PRESPER ECKERT, ENIAC co-inventor

TRACKING Moore's Law has been the goal of a major part of the processor industry during the last three decades. Until recently, the ever-increasing demand of computing power has been met on one hand by increasing the operating frequency of processors and on the other hand by designing architectures capable of exploiting parallelism at the instruction level through hardware mechanisms such as super-scalar execution. However, both these approaches seem to have reached a plateau, mainly due to issues related to design complexity and cost-effectiveness.

To face the stabilization of performance of single-threaded processors, the current trend in processor design seems to favor a switch to coarser-grain parallelization, typically at the thread level. In other words, high computational power is achieved not only by a single, very fast and very complex processor, but through the parallel operation of several processors, each executing a different thread. This kind of approach is currently implemented commercially through multi-core processors and in the research community through the *Multi-processors Systems On Chip* (MPSoC) approach, which is itself largely based on the *Network On Chip* (NoC) paradigm (see [Benini 02, de Micheli 06] or [Dally 01]).

Extrapolating this trend to take into account the vast amount of on-chip hardware resources that will be available in the next few decades (either through further shrinkage of silicon fabrication processes or by the introduction of molecular-scale devices), together with the predicted features of such devices (e.g., the impossibility of global synchronization or higher failure rates), it seems reasonable to foretell that current design techniques will not be able to cope with the requirements of next-generation electronic devices. Novel design tools and programming methods will have to be devised to cope with these requirements. The research presented in this thesis is an attempt to explore a possible avenue for the development of such tools.

1.1 Motivations and background

A tempting source of inspiration to solve the problems implied by a massively parallel organization and inherently error-prone substrates is biology. In fact, the complexity of living systems is apparent both in their sheer numbers of parts and in their behaviors: organisms are *robust* to damage and some can even *regenerate* (e.g. plants, hydra, salamanders); organisms exhibit dynamic characteristics (cells die and are replaced, proteins are synthesized and broken down), giving them enormous adaptive potential; organisms show developmental homeostasis (the ability to develop correctly despite assaults from the environment or genetic mutations) and developmental plasticity (the ability to make different and appropriate phenotypes to fit the environment).

Achieving at least a semblance of these properties is a challenge for current electronics systems. In fact, the application of bio-inspired mechanisms such as evolution, growth or self-repair in hardware requires resources (fault-detection logic, self-replication mechanisms, ...) that are normally not

available in traditional off-the-shelf circuits. For these reasons, over the past several years a number of dedicated hardware devices have been developed, such as the BioWall [Tempesti 01, Tempesti 03], the POETIC tissue [Thoma 04], or the PERPLEXUS platform [Upegui 07].

1.1.1 The bio-inspired approach

Biological inspiration has been used within these devices following a three tiered model called POE [Sanchez 96], which subdivides the various approaches in different axes:

1. The *phylogenetic* axis (P), which relates to the development of species through evolution.
2. The *ontogenetic* axis (O), which concerns the growth of organisms from a single cell to multicellular adults.
3. The *epigenetic* axis (E), which focuses on the influences of learning and of the environment on an organism.

The aforementioned devices have been successfully used to explore these various bio-inspired paradigms, but in general they represent experimental platforms that are very difficult to program and require an in-depth understanding of their underlying hardware. As a consequence, these platforms are accessible only to a limited class of programmers who are well-versed in hardware description languages (such as VHDL) and who are willing to invest considerable time in learning how to design hardware for a specific, often ill-documented device.

Notwithstanding these issues, hardware remains an interesting option in bio-inspired research domains as it can greatly accelerate some operations and because it allows a direct interaction with the environment. It is however undeniable that the difficulty of efficiently programming hardware platforms has prevented their use for complex real-world applications. In turn, the fact that experiments have been mostly limited to simple demonstrators has hindered the widespread acceptance of the bio-inspired techniques they were meant to illustrate.

Overall, these bio-inspired approaches seem very promising but their interest for a wider audience is problematic because their heavily hardware-oriented designs lack some of the flexibility achievable with a general purpose processor.

Thus, *the first objective* of this thesis will be the introduction of a processor-grade processing element at the heart of a bio-inspired hardware system. This processor needs to possess key capabilities, such as very good flexibility, to be able to maintain the versatility required by the implementation of bio-inspired mechanisms. Thus, the processing element we will propose will be able to fulfill the tasks traditionally involved in bio-inspired hardware but it will also enable a more traditional, software-oriented perspective, to realize general computation thanks to tools such as a compiler. The goal of this endeavor is to improve the appeal of bio-inspired approaches as a whole by showing how they can improve certain aspects of computation in real-world applications.

The *second objective* of this thesis will be to propose different hardware and software solutions to help use such a processing element in the context of *cellular computing*, a paradigm that resembles the cellular organization of living organisms.

1.1.2 Cellular architectures

While, by their very nature, systems that draw their inspiration from the multi-cellular structure of biological organisms rely on non-conventional mechanisms, in the majority of cases they bear some degree of similarity to networks of computational nodes, a structure that comes to resemble another computational paradigm, known as cellular computing.

Loosely based on the observation that biological organisms are in fact highly complex structures realized by the parallel operation of vast numbers of relatively simple elements (the cells), the *cellular computing* paradigm tries to draw an analogy between multi-cellular organisms and multi-processor systems. At the base of this analogy lies the observation that almost all living beings, with the notable exceptions of viruses and bacteria, share the same basic principles for their organization. Based on cell

differentiation, the incredible complexity present in organisms is based on a bottom-up self-assembly process where cells having a limited function achieve very complex behaviors by assembling into specific structures and operating in parallel. Thus, in the context of thread-level parallelism in a computing machine, a cellular architecture could be seen as a very large array of similar, relatively simple interconnected computing elements that execute in parallel the different parts of a given application.

Cellular architectures constitute a relatively recent paradigm in parallel computing. Pushing the limits of multi-core architectures to their logical conclusion where the programmer has the possibility to run many threads concurrently, this architecture is based on the duplication of many similar computing elements. Containing all the memory, computational power and communication capabilities, each cell provides a complete environment for running a whole thread. Thus, provided that network and memory resources are sufficient, the problem of achieving greater performance becomes a matter of how many cells can be put in parallel and, by extension, of how well thread-level parallelism can be extracted from the application. In practice, however, the cells interconnection network, i.e. the coupling between the cells, imposes limits both because of its topology and its technology.

Cellular organization also brings an interesting advantage in terms of design reuse and high testability: once a cell design has been tested thoroughly, it can be replicated as many times as the budget allows. Since the global structure relies on the same block, its assembly can be considered correct, interconnections apart. Another advantage implied by this organization is that faulty parts can be isolated easily and shut down if required.

Depending on the authors, the *cells* may comprise different levels of complexity ranging from very simple, locally-connected, logic elements to high-performance computing units endowed with memory and complex network capabilities. Thus, the actual interpretations and implementations of this paradigm are extremely varied, ranging from theoretical studies [Sipper 99, Sipper 00] to commercial realizations (notably, the *Cell* CPU [Pham 05, Pham 06] jointly developed by IBM, Sony and Toshiba), through wetware-based systems [Amos 04], OS-based mechanisms [Govil 99] and amorphous computing approaches [Abelson 00].

Even if the term of cellular computing regroups very different approaches, it always concerns systems in which a certain form of parallel computation can be performed. However, in and for itself, cellular computing leaves open the question how cells can be programmed. If adequately programming a parallel machine with two or four cores requires skills that not all programmers necessarily possess, with the advent of machines with 64 or even more cores this issue will become even more acute despite the non-negligible research efforts that have been undertaken to propose solutions to this question. It is therefore legitimate to wonder if the cellular approach can be applied to bio-inspired hardware systems as well, as a solution to allow researchers to rapidly prototype new ideas and, more importantly, to cope with the complexity of tens, hundreds, or thousands of parallel computational elements.

1.2 Objectives

As stated, the objectives of this thesis are two-fold: to propose a processor flexible enough to be tailored to different situations and then to propose solutions to use it easily in a parallel environment. Given the limitations of previous bio-inspired approaches to cellular computing, which require a considerable time investment to be used, another objective of this thesis is *to propose hardware and software solutions that enable novice users to easily harvest the computational power of massively parallel cellular systems*.

The central question of our thesis will then be to search if and how bio-inspired cellular architectures can be made accessible to a wider range of programmers while preserving their characteristics. This question will raise different issues, such as the impact of the implementation of this computing paradigm on performance but also the feasibility of its application to real programs.

Our work hypothesis is that with an adequate perspective on bio-inspired cellular architectures, which can be attained by different software abstractions, it becomes possible to limit some of the problematic characteristics (such as concurrency issues) of these parallel systems. However, our objective is not to replace the traditional approach of parallel programming but to propose an alternative vision of it in which bio-inspiration has a role to play, even if limited. The aim is thus to be able to easily apply

some bio-inspired mechanisms to widespread algorithms, such as AES encryption, to demonstrate the validity of the approach.

1.3 Thesis plan and contributions

Our thesis is articulated around two major parts. The first one consists of exploring how a little-known architecture, called *transport-triggered architecture* (TTA), enables on one hand to realize a complete general-purpose processor using a single instruction and, on the other hand, to demonstrate that it can be adapted to the needs of bio-inspired cellular systems. The second part leverages on the developments achieved during the first part by using the realized processor in a networked parallel prototyping platform, supported by a complete software framework that helps develop massively parallel applications.

1.3.1 Part 1: A highly customizable processing element

To bridge the gap between programmable logic and the kind of software tools required to implement real-world applications, we opted for *processor-scale computational elements* that provide an environment for running a thread of a distributed application. These elements also meet quite closely the requirements of the bio-inspired computing approach: substantially different from conventional computing units, these processors possess some key features that make them well-suited to implement the cells of our multi-cellular organisms.

To present this novel processor, we will proceed as follows: in chapter 2, we will first present an overview of some existing processor architectures and of the relevant features that led to the development of TTA. In the same chapter, we will also examine the strengths and weaknesses of the TTA approach in the particular context of this thesis.

The model introduced, we will focus in chapter 3 on the different alternatives that had to be considered during the *implementation* of the processor, named ULYSSE, so that it can fulfill all the requirements of the bio-inspired approach.

If the processor in itself presents interesting opportunities, notably in terms of reconfigurability and versatility as we will see, its programming model is too different from standard architectures to be easily programmed with a simple assembler. To limit the influence of the architecture on programming, we will present in chapter 4 two different software tools, consisting of a *macro assembler* and a *GCC-based compiler*. We will show that these tools enable the programmer to consider our TTA-based processor, depending on the context, either as a bio-inspired processor that possesses key capabilities in terms of reconfigurability or as a standard processor.

Chapter 5 will be dedicated to the *testing and performance evaluation* of the processor. Because of the importance of having a trustworthy platform, we will show that correct behavior is ensured by different levels of testing, ranging from low-level hardware assertions to an in-circuit debugger. After that, we will present the different *performance benchmarks* that were conducted to analyze the processor's performance on several typical embedded applications.

The processor and its programming environment complete, we will then show that its versatility makes it an interesting candidate for applying hardware-software co-design techniques to *evolve the structure of the processor itself*. Thus, in chapter 6, we will apply a novel partitioning technique based on genetic algorithms to determine which parts of a program are the most interesting candidates to be implemented in hardware. As we will see, the hybridization of genetic algorithms, which were used before in partitioning with only relative success , allows to obtain solutions very quickly.

1.3.2 Part 2: Networks of processing elements

The second part of this thesis is based on a multiprocessor platform, named CONFETTI, which consists of several reconfigurable FPGA circuits organized in a mesh topology. The flexibility of the platform allows the implementation of *arbitrary connection networks* for inter-processor transmissions, an invalu-

able capability to approximate the kind of highly-complex communication that enables biological cells to exchange information within an organism.

Notably, we will analyze the hardware realization of the platform itself in chapter 7 and then we will demonstrate how a complete routing system, based on the *globally-asynchronous, locally-synchronous* (GALS) paradigm [Teehan 07], can be implemented. After that, we will show how, in conjunction with the implementation of ULYSSE inside these FPGAs, different *software layers* can be applied to simplify the usage of the hardware platform.

These software layers are of different types: at low-level, they consist of a *communication library*, described in chapter 8, that can be used for instance to realize *cellular automata* on an asynchronous hardware substrate with only a few lines of C code.

At a higher level, presented in chapter 9, they consist of a *graphical user interface* that proposes an entire *design flow* for cellular applications that leads from application code, written in C, to a complete parallel system implemented on a hardware substrate. To guide and help developers during the realization of cellular applications, several tools are provided such as a profiler or two different simulators.

Before concluding, we will present in chapter 10 a dynamic distributed algorithm that enables applications to *grow and self-organize*, i.e. organize their topology according to the application requirements, on the cellular substrate of CONFETTI. More precisely, we will show how, when a computation node become overwhelmed by incoming jobs, it replicates on a different processor to balance the load. This algorithm, named *self-scaling stream processing* (SSSP), will then be validated with two different standard applications: AES encryption and MJPEG compression.

In the context of the design of bio-inspired hardware systems, we will demonstrate that two aspects of these tools are particularly useful. First, they are designed in such a way that it becomes relatively simple to introduce mechanisms such as learning, evolution, and development to any application. The second useful feature of the tools is that they are not, for the most part, tied to a hardware implementation: while we used the above-mentioned hardware setup in our experiments, most of the tools are quite general and can be applied to almost any network of computational nodes, whether they be conventional processors or dedicated elements. This flexibility comes from a decoupling between the hardware and the software layers, which allows the programmer to prototype bio-inspired approaches without necessarily knowing hardware description languages and specific implementation details.

Remark on bibliography *Each chapter of this thesis is directly followed by a bibliography that includes all the references cited within the chapter. As the subjects treated in each chapter vary greatly, this solution was adopted to allow the reader to find the references corresponding to a given subject more easily. In addition, all the references are repeated in a global bibliography in the appendix to provide a complete overview of the referenced work.*

Bibliography

[Abelson 00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman & Ron Weiss. *Amorphous computing*. Communications of the ACM, vol. 43, no. 5, pages 74–82, 2000.

[Amos 04] Martyn Amos. *Cellular computing*. Oxford University Press, New York, 2004.

[Benini 02] Luca Benini & Giovanni de Micheli. *Networks on Chips: A New SoC Paradigm*. Computer, vol. 35, no. 1, pages 70–78, 2002.

[Dally 01] William J. Dally & Brian Towles. *Route packets, not wires: on-chip inteconnection networks*. In DAC '01: Proc. 38th Conf. on Design automation, pages 684–689, New York, USA, 2001. ACM Press.

[de Micheli 06] Giovanni de Micheli & Luca Benini. *Networks on Chips: Technology and Tools (Systems on Silicon)*. Morgan Kaufmann, first edition, 2006.

[Govil 99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang & Mendel Rosenblum. *Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors*. In SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles, pages 154–169, New York, USA, 1999. ACM Press.

[Pham 05] D.C Pham, E. Behnen, M. Bolliger, H.P. Hostee, C. Johns, J. Kalhe, A. Kameyama & J. Keaty. *The design methodology and implementation of a first-generation CELL processor: a multi-core SoC*. In Proceedings of the Custom Integrated Circuits Conference, pages 45–49. IEEE Computer Society, September 2005.

[Pham 06] D.C Pham, T. Aipperspach & D. Boerstler et al. *Overview of the architecture, circuit design, and physical implementation of a first-generation CELL processor*. IEEE Solid-State Circuits, vol. 41, no. 1, pages 179–196, 2006.

[Sanchez 96] Eduardo Sanchez, Daniel Mange, Moshe Sipper, Marco Tomassini, Andrés Pérez-Uribe & André Stauffer. *Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware*. In Proceedings of the First International Conference on Evolvable Systems (ICES'96), pages 35–54, London, UK, 1996. Springer-Verlag.

[Sipper 99] Moshe Sipper. *The emergence of cellular computing*. Computer, vol. 32, no. 7, pages 18–26, July 1999.

[Sipper 00] Moshe Sipper & Eduardo Sanchez. *Configurable chips meld software and hardware*. Computer, vol. 33, no. 1, pages 120–121, January 2000.

[Teehan 07] Paul Teehan, Mark Greenstreet & Guy Lemieux. *A Survey and Taxonomy of GALS Design Styles*. IEEE Design and Test, vol. 24, no. 5, pages 418–428, 2007.

[Tempesti 01] Gianluca Tempesti, Daniel Mange, André Stauffer & Christof Teuscher. *The BioWall: an electronic tissue for prototyping bio-inspired systems*. In Proceedings of the 3rd Nasa/DoD Workshop on Evolvable Hardware, pages 185–192, Long Beach, California, July 2001. IEEE Computer Society.

[Tempesti 03] Gianluca Tempesti & Christof Teuscher. *Biology Goes Digital: An array of 5,700 Spartan FPGAs brings the BioWall to "life"*. XCell Journal, pages 40–45, Fall 2003.

[Thoma 04] Yann Thoma, Gianluca Tempesti, Eduardo Sanchez & J.-M. Moreno Arostegui. *POEtic: An Electronic Tissue for Bio-Inspired Cellular Applications*. BioSystems, vol. 74, pages 191–200, August 2004.

[Upegui 07] Andres Upegui, Yann Thoma, Eduardo Sanchez, Andres Perez-Uribe, Juan-Manuel Moreno Arostegui & Jordi Madrenas. *The Perplexus bio-inspired reconfigurable circuit*. In Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS07), pages 600–605, Washington, USA, 2007.

Part I

A highly customizable processing element

Chapter 2

Towards transport triggered architectures

"He who has not first laid his foundations may be able with great ability to lay them afterwards, but they will be laid with trouble to the architect and danger to the building."

NICCOLÒ MACHIAVELLI, *The Prince*

2.1 Introduction

PROCESSORS ARE ubiquitous in today's world. Although some of the most powerful of them act as central processing units in desktop PCs or servers, the great majority of processors are used in embedded systems. As stated in a recent issue of a specialized journal on microprocessors, *"consumer-electronics products are devouring embedded processors by the billions"* [Halfhill 08]. Thus, the mobile phones we use every day can easily contain two or three processor while high-end cars can have as much as fifty. In 2007, ARM shipped its ten-billionth core [Halfhill 08], a context that renders easily understandable the fact that every major player in the semiconductor industry proposes embedded processor in their catalog.

In this section, we will examine different programming models that exist among this plethora of embedded processors. We will begin with a brief historical overview that will enable us to distinguish three main tendencies in computer architectures. Starting with the *Reduced Instruction Set Computer* (RISC) model, we will then consider the VLIW (*Very Large Instruction Word*) model. This quick outline of various processor architectures will finally bring us to the *Transport-Triggered Architecture* (TTA) paradigm, the model that was used as the building block of our bio-inspired systems.

2.1.1 Preliminary note

The goal of the forthcoming section is to provide a context for the TTA approach. It should not be understood as an exhaustive description of every possible processor architecture through the ages, a role that documents such as [Hennessy 03] or [Patterson 98] fulfill extremely well, but more like as a basic overview.

In addition, the architecture descriptions that we are going to present will be limited to their very essence: in reality, every implementation of an architecture, understood as an interface between the hardware programmer and the software programmer, is sometimes forced to make concessions to the theoretical model in order to fit performance and size constraints imposed by the hardware.

2.2 On processor architectures

Historically, the 1980s and 1990s saw a "war" between two different approaches to realize microprocessors, approaches that differed most notably by their instruction set. One of these approaches was called CISC, an acronym that stands for *Complex Instruction Set Computer*. These architectures relied on a wide number of different instructions, using different formats, which required varying numbers of clock cycles to execute. Notable members of this family were DEC's VAX, Motorola's 68000 family and Intel's x86 processors (before the advent of Pentium processors). The other competing approach, RISC, counted among members processors such as the ARM, MIPS and SPARC.

When considering the embedded market, the outcome of this "war" is relatively clear:

> "In sheer volumes of 32- and 64-bit processors, RISC massacred CISC. For every PC or server processor that Intel sells, ARM's army of licensees sells five or ten ARM-based chips. Adding all the other RISC architectures – ARC, MIPS, SPARC, Tensilica, the Power Architecture, and more – makes the RISC victory look overwhelming. Furthermore, RISC processors rule the fastest-growing, most innovative markets. Mobile phones, iPods, TiVos, videogame consoles, portable video players, digital cameras, digital TVs, set-top boxes, DVD players, VoIP phones, flash-memory drives, memory cards, and numerous other products have RISC controllers. The x86 is found mostly in traditional PCs and servers." [Halfhill 08, p.11]

Because of the prevalence of RISC processors in the embedded field, we will now examine in more detail what makes the RISC processors better suited for the embedded market and, more specifically, we will examine what specificities could be interesting in the context of this research.

2.3 RISC architectures

The first ideas behind RISC date from 1964 when Seymour Cray was working on the CDC 6600 machine. During the late 1970s, these concepts were mainly developed in three different projects: the MIPS project at Stanford under the supervision of John Henessy, the RISC-I developed at Berkeley by David Patterson[1] and his team [Patterson 85] and the IBM 801 [Cocke 90].

In contradiction to the widely spread CISC processors available at that time which sought to achieve better performance by exploiting complex and powerful instruction sets, the RISC concept emphasized the insight that simple instructions could be used to provide higher performance in certain cases. Interestingly, a counter-argument against RISC machines was that more instructions were required to achieve the same goal when compared with a CISC implementation. Nevertheless, the RISC model was very prolific and showed high efficiency in many domains. Typical processor features that could provide the circumstances required for this to happen are:

- a lot of identical registers that could be used in any context;
- a reduced number of hardware supported data types (integer and floating point data);
- a fixed instruction size and unified instruction format, allowing a simpler decode logic;
- an ideal execution time of one cycle per instruction;
- a reduced number of different, simple instructions;
- a restricted amount of addressing modes;
- memory access only through the use of *load* and *store* operations, implying that normally operations are performed with register operands.

[1] The term RISC was invented for this project.

Of course, all these characteristics are implemented differently and to various extents in real processors and must be considered more as guidelines than strict requirements. As such, they must be more understood in comparison with their CISC alternatives (for instance, complex instructions working on string data types) than for themselves only.

2.3.1 Improving performance

Implementing the aforementioned guidelines provided some space for performance improvements. Notably, it became easier to design decode logic and more transistors could be used for the logic part of the CPU. This observation led, sometimes in contradiction with the initial aim of simplicity in RISC architectures, to the implementation of several solutions that span different levels of complexity.

Multiple opportunities for improvement appear with the following definition of *performance*:

$$\begin{aligned} \text{Performance} &= \frac{1}{\text{Execution time}} \\ &= \frac{f_{clock} \cdot IPC}{\text{Instruction count}} \\ &= \frac{f_{clock}}{\text{Instruction count} \cdot CPI} \end{aligned}$$

First, the clock rate of processors (f_{clock}) was increased every year, between 1995 and 2005, by about 30% per year [Flynn 99]. Besides this kind of strictly technology-based solutions such as accelerating the clock, increasing the performance of a processor was achieved by acting on the following factors:

- By using wider datapaths in processors (nowadays desktop processors generally use 64 bit general purpose registers).
- By reducing the transport penalty implied by memory access. This gave birth to a variety of *caching techniques*.
- By increasing the complexity of instructions. Even if this approach runs contrary to the RISC philosophy, sometimes it is of interest to increase the code density or when *customizable processors* are considered. This will be examined in detail in chapter 6.
- By decreasing the clocks per instruction (CPI), which could be done notably with efficient caching.
- By working on larger data words or on multiple data at the same time. This is the SIMD (*Single Instruction Multiple Data*) approach.
- By increasing the number of executed instructions per cycle (IPC). Widespread techniques to achieve this goal are *pipelining* and *superscalar execution*, which augment the number of instructions simultaneously executed by the processor.

2.4 VLIW architectures

If *instruction-level parallelism* (ILP), and hence performance, can be improved by superscalar execution, the technique is not without its problems. Resource allocation, extra pipeline stages but also dependencies checking and removal have a non-negligible cost when translated into the number of gates used in complex hardware design [Johnson 91]. A different approach is used to exploit ILP in *Very Large Instruction Word* (VLIW) processors: while many parallel execution units are also present, allocation and dependency check are done at compile time.

The model itself is relatively old and the first commercial implementations of it are the *Multiflow* [Colwell 88] and the *Cydra 5* [Rau 89] machines whereas more recent instances of the model can be found with the Philips *Trimedia* [van Eijndhoven 99] or Intel *Itanium 2* processors [McNairy 03,Huck 00].

ILP in the VLIW model is rendered explicitly visible to the compiler so it can exploit it as much as possible. Unlike in complex RISC processors, the work of each unit present in the processor is

analyzed at compile time and reordered in a way that tries to minimize the dependencies between instructions and maximize the occupation of each computing unit. Once the code has been compiled, it is then assembled in very long instructions that contain the task of every processing unit in the processor. Thus, even if VLIW processors possess computation capabilities similar to super-scalar RISC processors, their difference resides mainly in the fact that the programming model is parallel for the former and sequential for the latter.

The major consequence of this difference is that VLIW processors do not look for parallelism at run-time. Because this analysis has already been done at compile-time and also because the dependencies have already been checked, VLIW processors do not need to implement the relative hardware: complexity is reduced thanks to the removal of reordering, renaming and dependency check logic.

However, several conditions are required to allow this:

1. **Transparency** – Every register must be accessible to the compiler because a datum must be accessible from the moment of its creation in a functional unit until its replacement by another datum. For the same reason, register renaming is absent in this architecture.

2. **Known instruction latencies** – In order to statically manage the dependencies, the compiler must know the latency of every instruction to determine when a datum is "alive".

3. **Deterministic behavior** – As execution times must be known by the compiler, several standard optimizations have to be redesigned. For instance, speculative execution, which basically consists of executing an operation before knowing if the result will be effectively used, has to be setup in software. Another example is that in-order execution is mandatory.

The *explicitly-parallel instruction computing* (EPIC) model tried to answer the VLIW shortcomings by proposing solutions to the aforementioned problems. A well-known commercial implementation of this paradigm is the Intel's *Itanium* family whereas the *IMPACT* [Chang 91] or *PLAYDOH* [Schlansker 00] projects are illustrative research projects featuring EPIC architectures.

In summary, we could say that VLIW processors circumvented complex decoding logic and reordering mechanisms by proposing a more regular architecture which has both the advantages of being scalable and of enabling eased customization. Consequently, they are good candidates for developing high-performance embedded processors.

However, this came at the cost of binary compatibility, programs having to be recompiled to accommodate changes in the functional units. Additionally, they require adequate compiler support that took time to develop, the complexity of automatic parallelization of programs being huge. Finally, the complexity of real, implemented VLIW processors, resides in their datapath, which requires multiple bypass channels and heavy register files with a number of ports increasing with each functional unit implemented.

Nevertheless, the number of VLIW processors developed, notably for the embedded market, is a strong signal that they have a role to play. However, they have failed to replace general purpose processors in desktop computers. Does it mean that they are not a viable alternative, at least from a commercial point of view ? This is an ample question that the following opinion summarizes well and that will conclude our short survey on major existing computer architectures:

> "The VLIW did not really fail. It failed in general-purpose computing, because little parallelism is to be found in control-intensive code, but the VLIW is, and will continue to be, successful in data-intensive workloads.
>
> If the lesson learned is correct, we should not expect miracles from compilers, because if there is no parallelism, the best compiler in the world will not find it. The VLIW's broken piece is the program, the algorithm itself. We keep trying to make processors squeeze the last droplet of performance out of these programs and the compilers that create them." [Baron 08]

2.5 Transport triggered architectures

The trend that led to the development of VLIW processors was driven by the constant search for ILP increase. As can be seen in Figure 2.1, this paradigm shift also meant that an increasing responsibility was devoted to the static phase of the execution process: compilation.

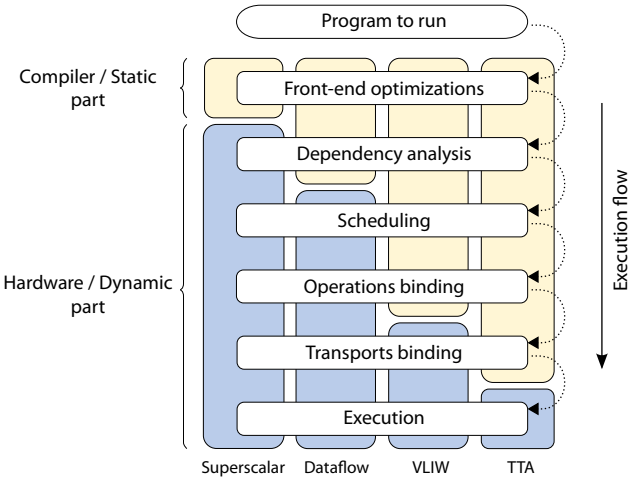


Figure 2.1: Separation of hardware-software responsibilities for some processor architectures.

As a result, today's compilers have to be able not only to produce optimized code but also to encompass a detailed vision of the targeted processor hardware model in order to be able to analyze dependencies between instructions and decide when and where to execute them. When moved to the static domain, the handling of these tasks becomes a complex matter, partly because the dynamic execution model of the processor should still be taken into account. In other words, as the information that is available at run-time is missing, it should be estimated, a process that constitutes a whole domain of research in itself and will not be discussed here.

Concurrently, the reduction in hardware complexity so obtained has the advantage of freeing some of the resources needed by prediction or dependency analysis, which in turn can be invested in improving the speed of computation.

2.5.1 Beyond VLIW

If this move to shift complexity from hardware to software was already well marked in VLIW and EPIC architectures, the opportunity to move yet more tasks to the software exists: in fact, the buses that transport the instructions inside the processor can also be made accessible to the programmer's model. The result of this shift is that every "traditional" instruction normally available in processors must now be considered as a sequence of one or more *data transfers*, computation being a "side-effect" of these transfers. This paradigm, which possesses several advantages in the context of this thesis (see section 2.6, will serve as a basis for the development of our bio-inspired processor and will be discussed in the remaining of this chapter.

The idea of triggering operations by data transports is relatively old. The first mentions we could find of such a computing paradigm come from middle of the seventies in the work of Lipovski and Tabak [Lipovski 75, Lipovski 77, Tabak 80]. After that, the idea was used in systolic arrays [Petkov 92], where computations are conducted as side-effects of data transports, in data-stream processors [Agerwala 82,Myers 81] and dataflow processors [Grafe 89,Dennis 88]. All those approaches can be regrouped under the term *data-centric* approaches because they have in common that they consider data, and not instructions, as the core of the computation.

2.5.2 Introducing the *Move*–TTA approach

More recently, the concept of triggering operations by data transfers was developed further with the *Move* paradigm, first described in the seminal work of Corporaal et al. at Delft University [Corporaal 97]. Originally intended for the design of application-specific dataflow processors, (processors where the instructions define the flow of data, rather than the operations to be executed) this paradigm must be considered as an extension of the VLIW model in which the operations are not central anymore. Thus, in the programming model proposed, there is a shift from the traditional *operation-triggered architectures* (OTAs) towards *transport-triggered architectures* (TTAs) that consider transport at the core of computation.

From an architectural perspective, rather than being structured, as is usual, around a more or less serial pipeline, processors designed following this model rely on a set of *functional units* (FUs) connected together by one or several *transport buses* that are used to carry parallel data transfers. The functional units realize all the different functions of the processor, be they arithmetic or I/O related, and the role of the instructions is simply to move data between the different FUs in the order required to implement the desired operations.

Since all the functional units are uniformly accessed through input and output registers, instruction decoding is reduced to its simplest expression, as only one instruction is needed: `move`.

Besides the theoretical analysis of the model itself, the work done at TU Delft also led to the creation a development framework called the *MOVE framework*[2] [Corporaal 91]. More recently, a similar work has been undertaken by Takala et al. at the Tampere University of Technology, under the label *TTA-based Codesign Environment* (TCE[3]) [Jääskeläinen 07]. Both frameworks propose sets of tools to explore, analyze and help with the hardware-software codesign of application-specific processors. In parallel of the development of these tools some PhD theses, notably by Hoffmann [Hoffmann 04], Hoogerbrugge [Hoogerbrugge 96] and Arnold [Arnold 01], have also been dedicated to the study of themes concerning the TTA approach or its applications.

2.5.3 From implicit to explicit specification

In many respects, the overall structure of a TTA-based system is fairly conventional: data and instructions are fetched to the processor from main memory using standard mechanisms (caches, memory management units, etc.) and are decoded as in conventional processors. The basic differences lay in the architecture of the processor itself, and hence in the instruction set.

Using a transport-centric approach means that TTA `move` instructions trigger operations that in fact correspond to normal RISC instructions. For example, a RISC `add` instruction specifies two operands and a result destination register that is updated with the sum of the operands at the end of the instruction. The *Move* paradigm requires a slightly different approach to obtain the same result:

```
// TTA equivalent of MIPS: addi $t0, 4, 3;
move adder_a, #3;
move adder_b, #4;
move t0, adder_result;
```

Instead of using a specific `add` instruction, the program moves the two operands to the input registers of a functional unit that implements the `add` operation. After a certain amount of time (the latency of the FUs), the result can then be retrieved from the output register of the functional unit and used wherever needed.

Instructions in TTA consist of three fields: two of them are used to store the source and destination addresses of the move operation whilst the last is used to indicate if the source field contains an immediate value or an address. As transfers are explicit in the architecture, nothings prevents the realization of parallel transfers using multiple transports buses, forming a special instance of the VLIW model in which every sub-instruction is a `move`. This situation is depicted in Figure 2.2 where simple `move` instructions are grouped together to form a VLIW-like instruction specifying multiple parallel

[2] `http://www.cs.tut.fi/~move/`
[3] `http://tce.cs.tut.fi/`

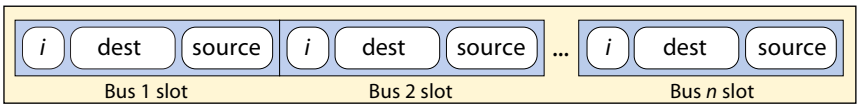


Figure 2.2: VLIW instruction word specification in TTA.

transfers. In this case not only must the data transports be rendered explicit, but also the buses to be used. Thus, almost every detail of the architecture is exposed and made available to the programmer's view, and hence the compiler.

2.5.4 Exposing various levels of connectivity

Figure 2.3, which is based on the same graphical representation used by Corporaal in [Corporaal 99], depicts a schematic view of an hypothetical TTA processor containing four transport buses and five functional units. The network controller that is present in the figure can be considered as the TTA equivalent of a fetch unit as it is in charge of controlling the pipelining of data transports between the FUs. These latter interface to the interconnection network using *sockets*, units that can possess either read or write ports. They respectively correspond to multiplexers and to demultiplexers that are used to put information to and from the FUs. Optionally, the FUs can implement a register in the interface endpoint with the sockets.

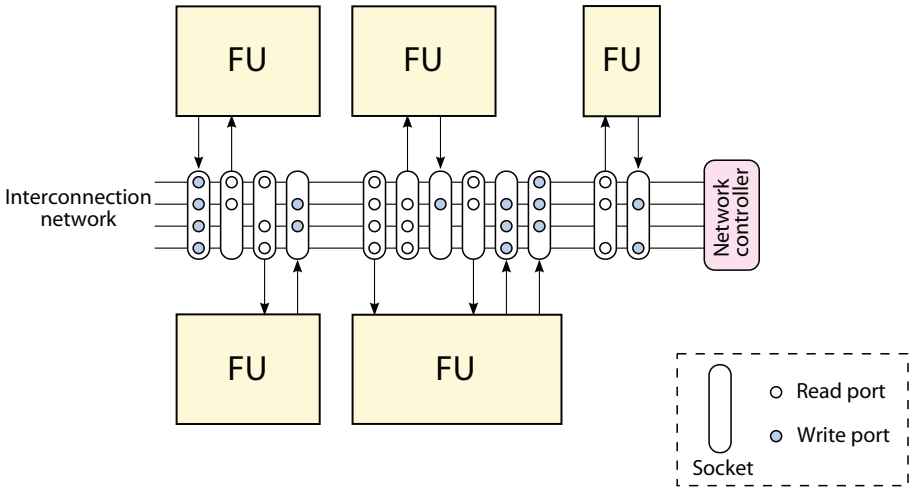


Figure 2.3: General architecture of a *TTA* processor.

Regarding the interconnection network, its functionality can be implemented in different ways as long as the displacements required by the application are allowed. When considering processors embedded in a single-chip configuration, it can be achieved using a traditional shared bus structure but nothing prevents implementations that specify transports outside the boundaries of the chip, as we will demonstrate in chapter 10.

However, if we limit our current discussion to single-chip configurations, connectivity can range from full-custom connectivity to full connectivity. In the first case, the interconnection network is tailored for a specific application: this is the case in Figure 2.3 where four concurrent transports are possible but only under certain specific circumstances. This yields a lightweight implementation along with a reduced bus load, at some expense in versatility. In case of full connectivity, the cost in terms of hardware is higher but code generation is simplified. We will analyze further this issue in section 4.2, when we consider the implementation challenges and opportunities that such an architecture poses for the development of a C compiler, but we can already mention that the task of the compiler in a TTA system consists of reducing the number of cycles for each data displacement, within the constraints of a given connectivity.

2.6 TTA advantages and disadvantages

If the main advantage of the TTA approach resides in its clear exposition of transport buses, it also enables specific optimizations. Notably, based on the observations that operations results are seldom used more than once and that they are often used immediately after their computation [Lozano 95, Martin 97, Cruz 00], it might be interesting in some cases to devise a new solution to reduce the importance of the monolithic structure that constitutes the *register file* (RF). This is particularly true in VLIW because of the inherent cost, in terms of material, to implement such structures when multiple accesses are required, as pointed out in [Corporaal 97, p. 102]:

> "VLIWs with K FUs need $3K$ ports on the RF: $2K$ read ports and K write ports[4]. These ports are needed because in the worst case situation each FU needs to perform two reads and one write on the RF simultaneously."

In TTA, because every displacement and location must be explicit, it is no longer required to structure registers in a file, nor to group arithmetic operators into an ALU. Thus, these structures can be distributed among the different FUs that compose the processor and we will expose notably in chapter 6, the advantages that can be drawn from such a situation. TTA processors offer thus the same flexibility as VLIW processors in the sense that FUs can be added or removed very easily. However, where VLIW processors have to face a dramatic increase in terms of complexity to implement the required RF ports and the bypass logic, TTA processors enable a different approach in which this issue is simplified.

2.6.1 Architecture strengths

The TTA approach, in and for itself, does not imply high performance. However, several arguments in its favor have been proposed, notably in [Corporaal 97, Nikhil 89, Hoogerbrugge 94] where the interested reader will find a more detailed perspective on the subject. In the context of this thesis, we retained the following pertinent elements:

1. **Modularity**
   From an architectural point-of-view, the *Move* model allows a distinct separation between the three main components at the core of the processor, namely:

   - *Computation*, with the use of distributed functional units instead of regrouped operators. Thus, arithmetic operators are not necessarily regrouped in an *arithmetic and logic unit* (ALU).
   - *Storage*, with registers not necessarily tied to register files.
   - *Communication*, with a flexible *routing infrastructure* instead of a rigid data path.

   In addition, each FU is connected to the other components of the system by means of a unified interface: the *socket*. From a designer's perspective, the modularity offered by this common interface is manifold.

   Firstly, it is interesting because it inherently implies that every component of a processor – functional units but also bus or memory interfaces – can be designed and tested separately as a distinct block. This allows a simplified generation of work units, which in turn simplifies work distribution in a team.

   Secondly, testability is also increased because each component is independent, which translates into the fact it can be fully and independently tested before being integrated with the rest of the processor.

   Thirdly, in the context of reconfigurable computing, it becomes possible to automate the design of processors by putting together different compatible building blocks that share a common interface.

[4] Assuming a VLIW with a shared RF, and FUs with two source and one destination operands.

A supplementary advantage of FU separation resides in the fact that FUs can possess multiple outputs and inputs, enabling any functionality to be implemented easily, even if it requires more than two inputs or produces more than one output.

Note that the idea of a *modular design* that uses various building blocks to create a processor has also been used in other architectures, such as [Rabaey 98] or [Hartenstein 99].

2. **Flexibility**
The explicit definition of the interconnection network that links the different FUs and the fact that only few constraints are imposed on its implementation allow a considerable amount of flexibility in its design. Notably, the architecture allows the number but also the nature (shared, external, ad-hoc, ...) of the buses used to transport instructions to be changed at will. Furthermore, as long as the implemented FUs adhere to the socket interface, there are no restrictions on how they are implemented. As a result, they can be easily pipelined and exhibit different latencies.

   Of course, VLIW and EPIC architectures also allow multiple operations to be performed in parallel but the addition or removal of any operation (which could roughly correspond to a TTA FU) requires the interconnection network and the register file to be changed accordingly. This process is much simpler with TTA processors thanks to a more marked separation between transport and computation.

3. **Scalability**
To keep the hardware footprint as low as possible, a TTA processor can be reduced to its minimal form because unused FUs can be easily removed (see section 3.1). Similarly, adding new FUs is also possible, for example to cope with special processing demands, such as a fast Fourier transform (FFT) unit.

   In addition, because only `move` operations have to be considered, changing the processor does not require the software tools, such as the assembler, to be changed to accommodate different hardware configurations. Furthermore, binary compatibility can be preserved between different CPU revisions, even when new FUs are added.

   We will see the full range of these advantages when we will consider the development of *ad-hoc* FUs in chapter 6.

4. **Efficiency**
From a strict hardware perspective, there are several advantages in implementing a TTA processor:

   - *Trivial decoding*. Because only a single instruction is present, decoding can be kept very simple and resources freed for other purposes.
   - *Fine-grained instruction level parallelism* (ILP) is achievable through VLIW-encoded instructions.
   - *Power savings*. In addition to the advantage, in terms of energy, of having a simple decoding logic, TTA processors can implement *clock gating* techniques to save power by turning off unused FUs. Moreover, the fact that the data transports are determined at compile time allows fine-grained power optimizations techniques to be used.
   - *Limited register file traffic*. This advantage comes from the fact that a register file is not strictly required, since a distributed version can be implemented by directly moving data from one FU to another, taking advantage of the fact that many produced results are used only once and hence do not need to be allocated to a general purpose register. The outcome of this distribution is that the traffic normally passing through the RF (which may constitute a bottleneck) is spread over the different FUs.
   - *Elimination of monolithic structures*. By splitting the register file and the ALU in different FUs, resource conflicts are reduced during instruction scheduling, which enables an increase in concurrency.

Software related advantages, implemented as compiler optimizations, will be discussed in detail in section 4.3.4.

5. **Hardware neutral perspective**
   The fact that the architecture handles the functional units as *"black boxes"*, i.e. without any inherent knowledge of their functionality, implies that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the addresses of the corresponding functional units.

   We define this hardware viewpoint as *hardware neutral* because in this situation, the hardware does not need to consider what the transports do, but only where to get and put the data, This is of particular interest for the development of the software tools (more on this in chapter 5).

2.6.2 Architecture weaknesses

Of course, the architecture also presents some drawbacks, which can be categorized into three different categories:

1. **Code complexity**
   The major problem with *Move* processors resides in the fact that programming such processors requires a completely different approach from any other architecture. The programmer's model that directly derives from the hardware consists only of `move` instructions, a narrow perspective that forces to think in terms of displacements and requires a long adaptation phase. In other words, because the programmer has to manage everything in terms of displacements, assembly programming becomes very difficult.

   Of course, the presence of a compiler solves this particular issue. The problem is that *Move* compilers are scarce, probably because of the complexity of generating efficient code in this context. In fact, *Move* processors compilers not only must face the same difficulties tackled as VLIW compilers, but must also be able to handle the specific optimizations allowed by the architecture.

2. **Expensive exception handling**
   If the disappearance of register files as monolithic structures presents some advantages, it renders exception handling and context saving very expensive. The reasons for this are simple: in the worst case, all the registers that are distributed among the different processor FUs must be saved and restored when an exception occurs. This could generate a prohibitive cost, notably because every register present must be made readable and writable to the architecture (hence, fully accessible on the buses). In addition, the problem of pipelined FUs (that present different latencies) must also be tackled when an exception or an interrupt occur.

   The combination of these two issues leads to problems with context switching, which is required for example by preemptive multitasking, and interrupts. This makes some people conclude that it is "currently considered not feasible" [Jääskeläinen 07] to port a multitasking OS for such an architecture. Yet, solutions exist (see for example [Corporaal 97, chapter 10]) to implement exceptions support in TTA processors, even if they can be quite expensive.

3. **Code size**
   Large instructions translate into large code, because the more parallel transport buses are present, the more data displacements must be specified. EPIC implementations solve the problem by compressing unused instruction slots that would have otherwise have been filled with `nop` operations.

   Similarly, code compression is applicable to TTA processors, with the advantage that it reduces memory needs and also energy consumption (see [Heikkinen 05]). Nevertheless, general code density is lower with *Move* processors than with traditional approaches [Myers 81, pp. 463–494] and [Corporaal 97, p. 243], which translates into higher memory requirements.

The *Move* model has, until now, been mainly restricted to the academic field. To our knowledge, the only notable industrial exception is the Maxim's *MaxQ* processor[5] [Maxim 04], which takes advantage of the clock-gating capabilities of the architecture to reduce power consumption to very low levels (the MAXQ200, a 16-bit microcontroller, can thus reach 3.5 MIPS/mA).

This relative lack of interest in the architecture is probably explained by the general absence of efficient and mature software tools that can tackle the programming difficulty of such processors and attain the performance of existing solutions.

2.7 A flexible operation-set

Aside from the advantages and disadvantages that have just been described and that aim at characterizing TTA processors from an architectural or implementation point-of-view, we consider that it is of particular interest to also examine TTAs from a programmer's perspective.

Even if the architecture uses only one kind of *instruction*, it is clear that TTA processors are able to perform more than one type of *operation*. In this context, an operation can be defined as:

Definition An *operation* is the set of modifications, in terms of the internal state of a processor, resulting from a single instruction.

Even for RISC processors, an instruction covers less than its corresponding operation. Let us consider the following MIPS instruction:

```
add r3, r1, r2;
```

This instruction has three effects. First, it changes the state of the `r3` register to correspond to the sum of the registers `r1` and `r2`. Second, the flag register of the processor is also updated to reflect potential overflows, the sign bit of the result, etc. Third, the PC value is updated to point to the next instruction. These three effects, following the aforementioned definition, correspond to the *operation* conducted. Keeping that in mind, it then becomes possible to define the *operation set* of a processor.

Definition The *operation set* of a processor is the set of every operation spanned by the instruction set of the processor.

In traditional architectures, the *operation set* does not play a very important role because it closely resembles the *instruction set*, the differences consisting mainly of the *side effects* of the instructions[6]. The situation is different for TTA architectures. In fact, from an architectural point-of-view, the situation can be seen as if such processors were programmed only with side effects.

Even if the instruction set of a TTA processor is reduced to its simplest expression (it is a singleton when not considering the parameters of the `move` instruction), its corresponding operation set spans the whole range of operations conducted by other processors. As such, TTA processors can also be considered as universal Turing machines that can carry any kind of computation using only one instruction.

In the context of this work, the existence of a dichotomy between instruction and operation set represents an undeniable advantage. In fact, this separation allows drastic changes in the processor *operation set* that would be much more difficult to obtain in every other architecture considered.

2.8 TTA and soft-cores

In parallel to all VLSI circuits, reconfigurable logic has also dramatically improved in the last years: FPGAs, for example, have become from a quantitative perspective, large enough to implement complete embedded processors. This has led to the appearance of a certain number of *soft-cores*, i.e.

[5] `http://www.maxim-ic.com/appnotes.cfm/appnote_number/3222`

[6] Such as flag settings or interrupt triggering.

processors that are proposed as software components that can then be realized into hardware within programmable logic circuits.

Some of these soft-cores are fully open-sourced, which is the case for example for the *Leon3-SPARC* processor[7], while some are only available as IP modules with a more limited configurability. In this last category, every major company in the reconfigurable logic industry has proposed specific processors (Microblaze™[8] for Xilinx, Nios™[9] for Altera, to cite only two). All these soft-cores share the capability to be configured to implement different modules such as UART communication modules, DMA or JTAG controllers, various memory interfaces.... In addition to these modules, which are often predefined, there also exists the possibility to implement VHDL components that can then be used by the processor to generate *ad-hoc* processors. Inside this plethora of possibilities, some researchers even propose *dynamically* reconfigurable processing elements, for example to accelerate some kinds of computation [Campi 07, Mucci 08].

The aforementioned processors are generally based on a RISC instruction set and access the various peripherals through a dedicated memory zone. One advantage of TTA processors in this context is that, because their whole operation is based on a memory map, i.e. each operation is implemented in an FU and accessible via a data displacement inside the memory space of the CPU, the architecture is completely *regular* and does not necessitate special mechanisms to implement an arbitrary number of FUs. As such, they are very good candidates for a configurable soft-core implementation.

Of course, from an hardware point of view, it is also possible to use the same memory mapping mechanism to perform operations in a RISC or VLIW architecture. However, the flexibility attained in the TTA approach, where the functional units are really *pluggable*, is undeniable, as we will see in the next chapters.

2.9 Conclusion

After having described the general strengths and weaknesses of TTA processors, it should be fairly obvious that they possess key capabilities in the context of our bio-inspired approach. First, the versatility of the TTA paradigm fulfills all the requirements for the synthesis of ontogenetic, application specific instruction set processors (ASIPs), that can assemble from basic building blocks. More specifically, the possibility of specializing the operation set with *ad-hoc* functional units enables to *adapt the processor to the application* while keeping the overall structure of the processor – fetch and decode unit, bus structure, etc. – unchanged. Moreover, the fact that architecturally only *data transports* are visible separates functional units (where computation is done) from their scheduling (when the computation is done).

[7] `http://www.gaisler.com`
[8] `http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm`
[9] `http://www.altera.com/products/ip/processors/nios/nio-index.html`

Bibliography

[Agerwala 82] T. Agerwala. *Data Flow Systems: Guest Editors' Introduction*. Computer, vol. 15, no. 2, pages 10–13, 1982.

[Arnold 01] Marnix Arnold. *Instruction set extension for embedded processors*. PhD thesis, Delft University of Technology, 2001.

[Baron 08] Max Baron. *VLIW: Failure or Lesson ?* Microprocessor Report, June 2008.

[Campi 07] Fabio Campi, Antonio Deledda, Matteo Pizzotti, Luca Ciccarelli, Pierluigi Rolandi, Claudio Mucci, Andrea Lodi, Arseni Vitkovski & Luca Vanzolini. *A dynamically adaptive DSP for heterogeneous reconfigurable platforms*. In Proceedings of the conference on Design, automation and test in Europe (DATE'07), pages 9–14, San Jose, CA, USA, 2007. EDA Consortium.

[Chang 91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter & Wen mei W. Hwu. *IMPACT: an architectural framework for multiple-instruction-issue processors*. SIGARCH Computer Architecture News, vol. 19, no. 3, pages 266–275, 1991.

[Cocke 90] John Cocke & Victoria Markstein. *The evolution of RISC technology at IBM*. IBM Journal of Research and Development, vol. 34, no. 1, pages 4–11, 1990.

[Colwell 88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth & Paul K. Rodman. *A VLIW architecture for a trace scheduling compiler*. IEEE Transactions on Computers, vol. 37, no. 8, pages 967–979, 1988.

[Corporaal 91] Henk Corporaal & Hans (J.M.) Mulder. *MOVE: a framework for high-performance processor design*. In Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 692–701, New York, USA, 1991. ACM.

[Corporaal 97] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, USA, 1997.

[Corporaal 99] Henk Corporaal. *TTAs: missing the ILP complexity wall*. Journal of Systems Architecture, vol. 45, pages 949–973, 1999.

[Cruz 00] José-Lorenzo Cruz, Antonio González, Mateo Valero & Nigel P. Topham. *Multiple-banked register file architectures*. SIGARCH Computer Architecture News, vol. 28, no. 2, pages 316–325, 2000.

[Dennis 88] J. B. Dennis & G. R. Gao. *An efficient pipelined dataflow processor architecture*. In Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Flynn 99] Michael J. Flynn, Patrick Hung & Kevin W. Rudd. *Deep-Submicron Microprocessor Design Issues*. IEEE Micro, vol. 19, no. 4, pages 11–22, 1999.

[Grafe 89] V. G. Grafe, G. S. Davidson, J. E. Hoch & V. P. Holmes. *The Epsilon dataflow processor*. SIGARCH Computer Architecture News, vol. 17, no. 3, pages 36–45, 1989.

[Halfhill 08] Tom R. Halfhill. *Intel's Tiny Atom – New low-power microarchitecture rejuvenates the embedded x86*. Microprocessor Report, vol. 1, pages 1–13, April 2008.

[Hartenstein 99] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann & Ulrich Nageldinger. *Mapping Applications onto Reconfigurable Kress Arrays*. In FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, pages 385–390, London, UK, 1999. Springer-Verlag.

[Heikkinen 05] J. Heikkinen, A. Cilio, J. Takala & H. Corporaal. *Dictionary-based program compression on transport triggered architectures*. In IEEE International Symposium on Circuits and Systems (ISCAS'2005), vol. 2, pages 1122–1125, May 2005.

[Hennessy 03] John L. Hennessy & David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[Hoffmann 04] Ralph Hoffmann. *Processeur à haut degré de parallélisme basé sur des composantes sérielles*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[Hoogerbrugge 94] Jan Hoogerbrugge & Henk Corporaal. *Transport-Triggering vs. Operation-Triggering*. In Proceedings of the 5th International Conference Compiler Construction, pages 435–449, January 1994.

[Hoogerbrugge 96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[Huck 00] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder & Rumi Zahir. *Introducing the IA-64 Architecture*. IEEE Micro, vol. 20, no. 5, pages 12–23, 2000.

[Jääskeläinen 07] Pekka Jääskeläinen, Vladimír Guzma, Andrea Cilio, Teemu Pitkänen & Jarmo Takala. *Codesign toolset for application-specific instruction-set processors*. In Reiner Creutzburg, Jarmo Takala & Jianfei Cai, editors, Proceedings SPIE Multimedia on Mobile Devices 2007, vol. 6507. SPIE, February 2007.

[Johnson 91] William M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[Lipovski 75] G. Jack Lipovski & J.A. Anderson. *A Micronetwork for Resource Sharing*. In Microarchitecture of Computer Systems, page 223, Nice, France, 1975.

[Lipovski 77] G. Jack Lipovski. *On virtual memories and micronetworks*. SIGARCH Computer Architecture News, vol. 5, no. 7, pages 125–134, 1977.

[Lozano 95] Luis A. Lozano & Guang R. Gao. *Exploiting short-lived variables in superscalar processors*. In Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28), pages 292–302, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[Martin 97] Milo M. Martin, Amir Roth & Charles N. Fischer. *Exploiting dead value information*. In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30), pages 125–135, Washington, USA, 1997. IEEE Computer Society.

[Maxim 04] Maxim. *Introduction to the MAXQ Architecture, Application Note 3222*. `http://pdfserv.maxim-ic.com/en/an/AN3222.pdf`, 2004.

[McNairy 03] Cameron McNairy & Don Soltis. *Itanium 2 Processor Microarchitecture*. IEEE Micro, vol. 23, no. 2, pages 44–55, 2003.

[Mucci 08] Claudio Mucci, Luca Vanzolini, Ilario Mirimin, Daniele Gazzola, Antonio Deledda, Sebastian Goller, Joachim Knaeblein, Axel Schneider, Luca Ciccarelli & Fabio Campi. *Implementation of parallel LFSR-based applications on an adaptive DSP featuring a pipelined configurable Gate Array*. In Proceedings of the conference on Design, automation and test in Europe (DATE'08), pages 1444–1449, New York, USA, 2008. ACM.

[Myers 81] Glenford J. Myers. *Advances in Computer Architectures*. Wiley-Interscience, 1981.

[Nikhil 89] R. S. Nikhil. *Can dataflow subsume von Neumann computing?* SIGARCH Computer Architecture News, vol. 17, no. 3, pages 262–272, 1989.

[Patterson 85] David A. Patterson. *Reduced instruction set computers*. Communications of the ACM, vol. 28, no. 1, pages 8–21, January 1985.

[Patterson 98] David A. Patterson & John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, second edition, 1998.

[Petkov 92] N. Petkov. *Systolic Parallel Processing*. Elsevier Science Inc., New York, USA, 1992.

[Rabaey 98] J. Rabaey & M. Wan. *An energy-conscious exploration methodology for reconfigurable DSPs*. In Proceedings of the conference on Design, automation and test in Europe (DATE'98), pages 341–342, Washington, USA, 1998. IEEE Computer Society.

[Rau 89] B. Ramakrishna Rau, David W.L. Yen & Ross A. Towle. *The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs*. IEEE Computer, vol. 22, no. 1, pages 12–35, 1989.

[Schlansker 00] Michael S. Schlansker & B. Ramakrishna Rau. *EPIC: Explicitly Parallel Instruction Computing*. Computer, vol. 33, no. 2, pages 37–45, 2000.

[Tabak 80] Daniel Tabak & G. Jack Lipovski. *MOVE Architecture in Digital Controllers*. IEEE Transactions on Computers, vol. 29, no. 2, pages 180–190, 1980.

[van Eijndhoven 99] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel & H. P. E. Vranken. *TriMedia CPU64 Architecture*. In Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers and Processors (ICCD'99), 1999.

Chapter 3

The ULYSSE processor

"Heureux qui, comme Ulysse, a fait un beau voyage,
Ou comme cestuy-là qui conquit la toison,
Et puis est retourné, plein d'usage et raison,
Vivre entre ses parents le reste de son âge !"

JOACHIM DU BELLAY, *Les Regrets*

TO EVALUATE the different trade-offs, design decisions, but also possible opportunities in terms of bio-inspiration, we have implemented a "soft-core" processor, named ULYSSE, that will be used as a cornerstone of this thesis. An instance of the TTA model, ULYSSE implements transport triggering along with different mechanisms to achieve a maximum flexibility along with a good level of performance.

In this chapter, we will describe the general architecture of this processor and also discuss the design decisions taken during the implementation. The software aspects being discussed separately in the next chapter together with the performance analysis of the resulting processor, this chapter will be devoted only to hardware-related issues. When pertinent, we will also compare our design decisions with the *MOVE32INT* processor [Corporaal 93], the only other non-commercial instance of *Move* processor that we are aware of.

Thus, the topics described in this chapter are the following: the chosen transport model (how data are transported inside the CPU), how pipelining works at the transport level, and also at the functional unit level. In addition to the operation of the processor itself, we will cover as well some interesting FUs that were developed in the context of this thesis. Finally, we will end our discussion regarding the hardware architecture with an analysis of further extensions that could be applied to the model.

3.1 Internal structure overview

ULYSSE is a 32-bit *soft* processor that follows the TTA approach. As such, it can be described in its essence as a set of FUs linked together with an interconnection network. To implement such a processor, three main classes of components are required:

1. **Interconnection network**
   It connects the different FUs and allows the data to be moved from one point to another inside the processor.

   Its realization details will be described in section 3.4.

2. **Fetch unit**
   It constitutes the entry point of the instructions inside the processor: it is in the fetch unit that the instructions are decoded and where reside the registers that permit to determine and change the flow of the program, namely the program counter (PC) and the instruction register (IR).

We will see in section 3.6 how it was realized and in section 3.8.2 how conditional jumps are handled within the processor.

3. **Functional units**
   They determine the operation set of the processor by implementing not only arithmetic operations but also more specific functionalities such as I/O operations[1], external memory interface or general purpose registers. All these FUs have in common their access method that consists of addressable locations that may optionally contain a register.

   We will detail, starting in section 3.7, the various FUs that we have developed in the context of this thesis.

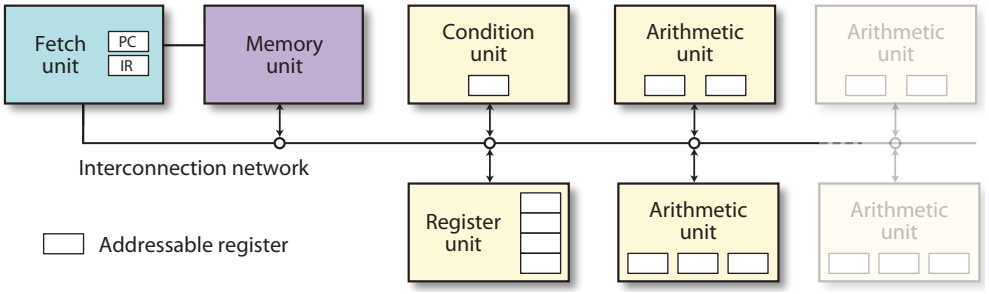


Figure 3.1: Overview of a minimal ULYSSE processor.

Thus, a schematic, minimal version of ULYSSE processor (depicted in Figure 3.1) is composed of a fetch unit, a memory interface, some arithmetic and logic functional units, a condition unit and an interconnection network to connect everything together. Depending on the application, it can also include a register unit containing only registers.

3.2 Simplified operation

As most of the hardware complexity has been transferred to the software side, the basic operation of the processor is quite simple. The *fetch unit* extracts from memory the instruction pointed to by the program counter and, after a trivial decoding phase, the address of the location serving as a source and the address of the destination location are placed on the interconnection network on two separate buses.

Every FU possesses an *internal address* (in what we call the *inner addressing space* of the processor) that is permanently matched with the source and destination buses. When a match occurs, the FU knows that it is used in an instruction. In that event, it writes or reads the data bus (which is 32 bits wide) depending on the instruction. In the case of immediate values, the data bus is assigned to the fetch unit that extracts immediate values directly from the instruction. By replicating the mechanism several times over, it is possible to realize multiple displacements in parallel.

3.2.1 Operation example

Let us now examine the mechanisms that are used in a very simple operation consisting of summing two numbers, 5 and 4, and then accumulating the result with 5 twice. The corresponding RISC code is as follows:

```
addi $t0, zero, 5;
addi $t0, $t0, 4;
addi $t0, $t0, 5;
addi $t0, $t0, 5;
```

[1] Such as network interface, display controller, general-purpose I/O pins (GPIOs), ...

Obtaining the same result with our processor requires a slightly different sequence of instructions. First, the operands to be added should be displaced to an adder FU which, after a certain amount of time (the latency of the operator, noted λ, in clock cycles units), will make the result available at one of its output. The result can then be retrieved and reused as an input of the adder FU the required number of times. The corresponding code is as follows:

```
move adderA, #5;
move adderB_t, #4;
move adderB_t, adder_r;
move adderB_t, adder_r;
```

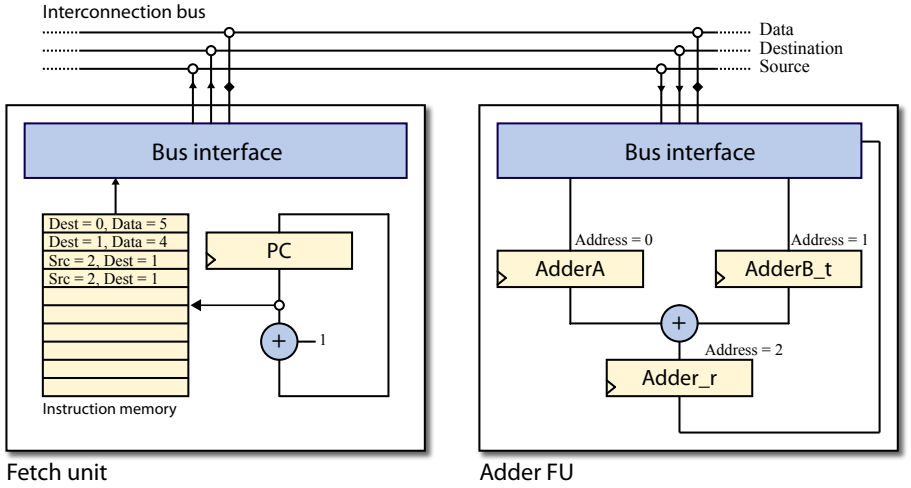


Figure 3.2: Schematic TTA adder example.

The operation of the program is represented in Figure 3.2, which depicts how the instructions, which correspond to displacements addresses, are fetched from a memory before being put on dedicated buses. In this example, the first `move` specifies to transfer the immediate value 5 to address 0, which is the *AdderA* register. The second displacement indicates that the immediate value 4 must be put into address 1 (register *AdderB_t*): here the "_t" suffix indicates that writing to this register *triggers* the operation (see next section), which in this case signifies that the value 9 will be put into register *Adder_r*. At the end of the execution of this program, *Adder_r* will store the value 19.

Several remarks can be made concerning this program. We can notably notice that it was possible to displace the value from the output register to the operand register without going through an intermediary one. Moreover, the final result remains in the output register of the adder unit until a new operation is triggered, which in this situation corresponds to writing to a special register. Other schemes are possible as we will discuss in the next section.

3.2.2 Triggering schemes

If data transports trigger operations in the chosen architecture, several triggering schemes can be used. In other words, different techniques can be used to determine *when* operations occur following a data transfer.

Hoffmann's taxonomy of triggering schemes [Hoffmann 04, p. 87–88] distinguishes several policies that define under which conditions an operator should start a computation and under which conditions a result should be produced. Besides the concept of *triggering operand* defined by Corporaal in [Corporaal 97], he defines the concept of *neutral operand* as an operand that does not produce visible results[2]. Such operands are rather rare and he cites the example of the data memory where the address

[2]The original text is "C'est l'absence de résultat visible qui justifie le qualificatif de neutre." [Hoffmann 04, p. 87].

to access the data is first stored before another input is used to define the operation to be carried out.

In total, we distinguish at least four different triggering schemes:

1. **Co-triggering**
   The FU realizes its corresponding operation when all the required operands have been updated.

2. **Triggering operand**
   The update of specific operands (which can be unique or not) triggers an operation, regardless of the state of the other operands. This triggering scheme is useful for instance to be able to modifiy non-triggering operands whilst keeping the output valid. This is the scheme used in the example of the previous section.

3. **Triggering input**
   An input of the FU is used only to trigger operations. This method is quite similar to the second but allows a bit more flexibility because every input operand is then equivalent (with no side-effect).

4. **Triggering read**
   An operation can be triggered when a particular output (or input) of an FU is *read*. This can be used for example to realize a FIFO memory FU, where a value is popped everytime it is read.

These different schemes are not exclusive and can be mixed together in a CPU to accommodate various needs. In the described processor, the *triggering operand* is almost always used.

3.3 Different CPU versions

At the time of writing this thesis, three different version of ULYSSE have been implemented on three different hardware platforms. All share the general TTA organization that was presented before and their main differences reside in the peripherals they contain, notably due to the requirements of the different hardware platforms targeted.

Historically, the first version developed targeted a PCI board hosted on a PC computer, on a Xilinx® VIRTEX™ II-3000 FPGA. Because of the relative large size of that FPGA, this first version makes use of embedded memory and can realize two `move` operations in parallel. For this reason, it is called the *VLIW* version. Despite this unique feature, this version is of little interest because its development was completely stopped at an early stage for reasons that will be explained in section 3.4.1.

The second version of the processor is a *standalone* version and is depicted in Figure 3.3, which details how the different components of the processor are interconnected but also how it interfaces with the outside world. This version of the processor targeted small FPGAs and for this reason it uses a SRAM controller along with a serial code-loader. It can also include one or several general purpose I/O (GPIO) controllers to interact with external hardware. Mainly developed as a debugging platform, it was implemented on a Xilinx® SPARTAN® 3 3XCS200 FGPA board.

The last version is called the *cellular* version because it targeted our cellular hardware substrate, discussed later in chapter 7. It is based on the *standalone* version but with the additional constraint that it is used in a multiprocessor environment in which it must communicate using a high-speed network interface that also serves to update the CPU code. Additionally, it also possesses an interface to a RGB led display. It is the most complete and most tested version of ULYSSE and, unless stated otherwise, we mainly discuss this version even if it is somewhat interchangeable with the *standalone* version.

3.4 Interconnection network

ULYSSE is programmed using only data transports from FU to FU. They take place on the *interconnection network* of the processor. Among the different compromises between performance and size that are available to implement the interconnection network, ULYSSE uses a shared bus topology with full connectivity, on which FUs are simply plugged in. This means that the interconnection network is

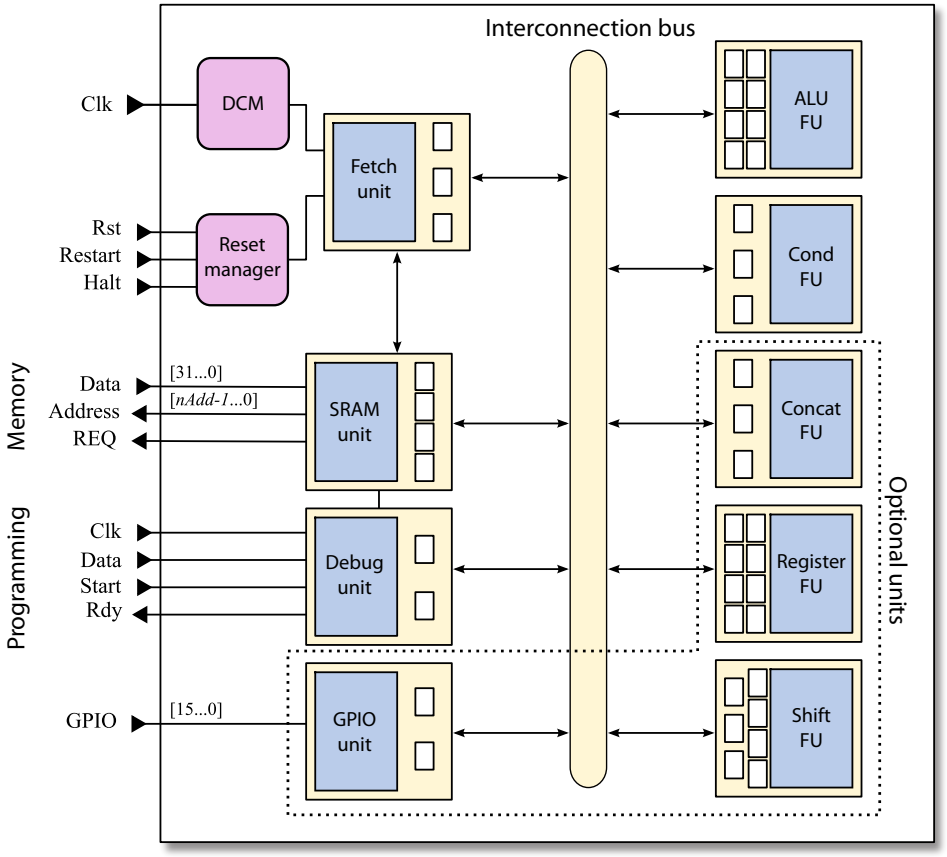


Figure 3.3: Detailed view of the ULYSSE processor in its *standalone* version.

composed of three different buses that contain the source address, the destination address and the datum to be transported. This network appears in Figure 3.2 or in Figure 3.6.

Even if this kind of connectivity is more expensive from an hardware perspective, it remains relatively simple to implement whilst proposing an adequate performance. Scheduling is also simplified because full connectivity enables displacements from any register to any other register in the CPU. Consequently, code generation is simpler.

In the *MOVE32INT* processor, a customizable version of the interconnection network was implemented, using *sockets* to add or remove connections from a particular FU to a transport bus. By analyzing different connectivity configurations, the influence of the connections on various aspects such as processor performance or size could be determined.

Another difference between the two processors lies in the number of parallel transport buses. In *MOVE32INT* processor, up to four data displacements in parallel can be specified whereas only two are possible in our case with the VLIW version.

3.4.1 Single slot instructions

The use of multiple slots instructions as proposed in the VLIW version was abandoned in the two major implementations – standalone and cellular – we are discussing here. The decision to focus on the implementation of a single `move` slot per instruction was made for several reasons:

Difficult programming in the absence of an adequate compiler The ULYSSE processor development started several years ago, at the very moment we began working on this thesis. At that time, several questions were still open. Notably, the existence of a valid compiler for that newborn processor was not certain. Consequently, we decided not to rely too much on this hypothetical

development but we left, as much as possible, the door open to future developments. Thus, a working version of ULYSSE with parallel displacements was developed but the advantages were only slim compared to a non-parallel version, due to the lack of an adequate compiler but also because assembler programming by hand was too difficult. Two main reasons explain this:

- Dependencies checking between the instructions is very difficult to do by hand and is the source of many errors.
- Writing an optimized program for multiple displacements (i.e. VLIW instructions) requires two passes on the code, one for writing what we want the processor to do followed by a second where the instructions are moved to free slots. The resulting code becomes then very difficult to read because pieces of code that should normally be together are disseminated in different slots.

Because of these reasons and the fact that enabling multiple parallel displacements was not our top priority, we decided at that time not to continue in that direction.

During the following years it turned out that, with the help of Michel Ganguin who realized his master thesis under my supervision, a working version of GCC was made available for this processor (see section 4.2). Of course, due to the huge complexity of such a piece of software, some bugs are still present but they remain completely bearable under the assumption that it is still under ongoing development. However, even if a basic compiler exists, optimizing it so it would be able to use the parallel displacement slots in an intelligent way goes beyond the subject of this thesis. In fact, the subject was treated in detail by Hoogerbrugge in his PhD dissertation [Hoogerbrugge 96].

Reduced memory size Because the target processing elements we focus our attention on are small-sized, embedded processors, the amount of memory they possess remains relatively modest. Considering the fact that *Move* code is already less dense than standard code (see [Myers 81, pp. 463–494] and [Corporaal 97, p. 243]), it is not realistic to implement enough slots so that VLIW can be interesting from a performance perspective. Moreover, more slots would even more complicate the situation explained in the previous point.

nop instructions and wasted memory The aforementioned limited memory problem also implies that space can not be wasted by the insertion of `nop` instructions to fill the unused slots, which would be necessarily the case without optimized and automatic methods to fill them at compile time. Of course, solutions exists to compress VLIW instructions (for instance [Xie 01, Lin 04]), but their implementation would not change anything to the problem of the lack of an efficient compiler.

3.4.2 Instruction format and inner addressing

One of the advantage of the *Move* architecture is that it does not necessitate complex decoding because only one instruction exists, with two variants:

1. `move` *address*, *address*
2. `move` *address*, *immediate value*

The `nop` "instruction" can be added to these two instructions by defining it as a displacement from address 0 to address 0. This formal definition can also be extended by noting that any displacement to a non-existing address also constitutes a `nop` instruction, a fact that we will use notably for testing (see section 5.1).

In summary, each instruction must code the following elements:

- a source address;
- a destination address;

- a flag to indicate if an instruction contains an immediate value or two addresses.

In the standalone and cellular implementations, immediate values are 16 bits[3] wide whereas source and destination addresses are coded with 15 bits each[4]. The immediate flag uses one bit. Thus, the specification of one displacement requires 32 bits with an immediate value and 31 for register-to-register moves. In the latter case, one bit of padding is used to align on the 32 bits boundary word. This gives the following instruction format (P being the padding bit):

$$\text{Immediate} \rightarrow \text{Address} \quad : \quad 1 \mid \underbrace{\texttt{DDDD...DDDD}}_{\text{15 bits dest.}} \mid \underbrace{\texttt{IIII...IIII}}_{\text{16 bits imm. value}} \tag{3.1}$$

$$\text{Address} \rightarrow \text{Address} \quad : \quad 0 \mid \underbrace{\texttt{DDDD...DDDD}}_{\text{15 bits dest.}} \mid \underbrace{\texttt{SSSS...SSSS}}_{\text{15 bits source}} \mid P \tag{3.2}$$

The *inner addressing space* (the space that can be addressed with these 15 bits) has been separated into 1024 segments (one per functional unit), each permitting 32 addresses that correspond to the maximal number of interfaced register per functional unit. This paginated access was implemented in order to reduce the complexity of the bus interface (see section 3.5) and yields to the following paginated representation:

$$(base, page) \quad \text{with } base \in [0, 1023] \text{ and } page \in [0, 31]$$

3.5 A common bus interface

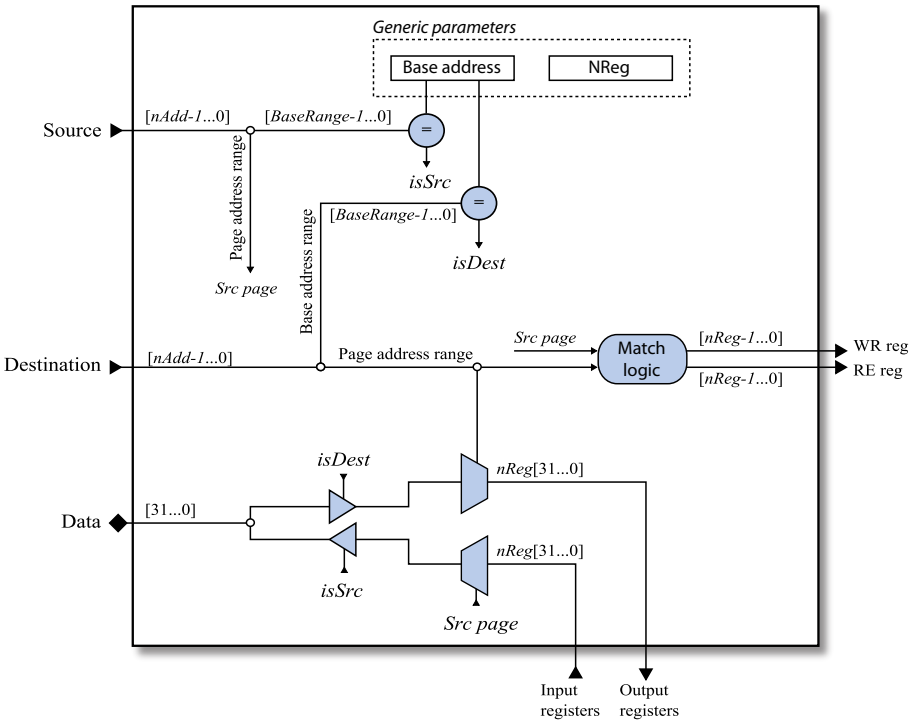


Figure 3.4: Schematic of the common bus interface.

[3]We will discuss in section 3.8.1 how immediate values that do not fit this range can be used.

[4]The VLIW implementation uses shorter inner addressing space with only 9 bits for immediate values and 8 bits for source and destination.

An important building block of the processor is the *bus interface* (Figure 3.4), or socket, that connects every FU to the interconnection network of the processor.

In our architecture, one important feature the bus interface had to possess is *scalability*, in the sense that the number of input or output registers had to be changed easily to accommodate different needs within the FU. By developing a standardized interface (the VHDL interface is reported in the annex, section A.1), we managed to achieve independence between the function realized and its interface within TTA. Using the interface, any VHDL component can be easily integrated inside ULYSSE: it suffices to instantiate the bus interface and specify the number of interface ports (or registers) used to connect to the system[5].

The interface shown in the figure also provides signals (`re_regs` and `we_regs`) that indicate, using a *one-hot* bit encoding, when a register is accessed for reading or writing. This allows for instance to implement complex triggering schemes based on the fact that a particular location has been read or written.

The interest of using a paginated addressing scheme appears clearly: thanks to that, the comparison with the FU address uses only $BaseRange$ bits (i.e. 10 bits in the two main implementations).

Annexed listing (Listing A.2, p. 210) provides an example of a very simple FU – used to concatenate two immediate, 16 bit values to form a complete 32 bit word – that depicts the basic interface and how it is used in a real example to provide a simple, yet powerful interface between ULYSSE and its functional units.

3.6 The fetch unit

The primary role of the ULYSSE processor's fetch unit is to schedule the global operation of the processor, which is summarized by its state machine (Figure 3.5).

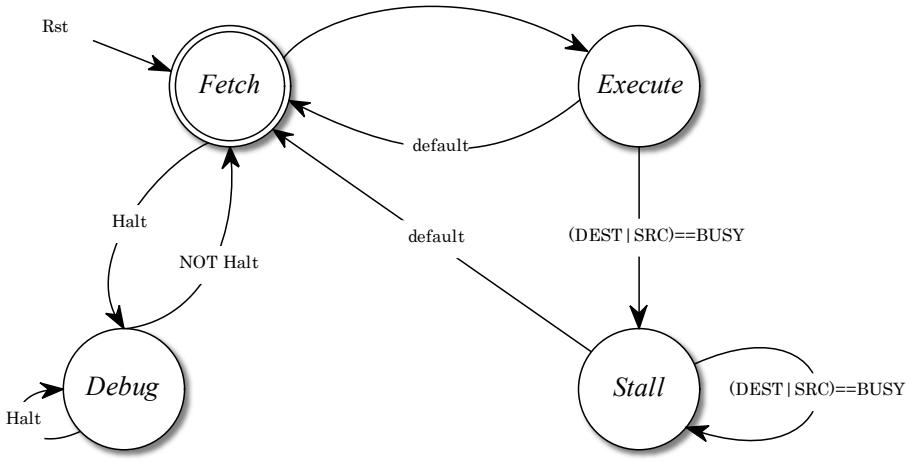


Figure 3.5: Fetch unit state machine.

The fetch unit realizes three main operations:

1. **Instruction fetching**
   The fetch unit is primarily a *load unit* which is responsible of requesting new instructions from the memory. To fulfill this task, it does not include directly a memory controller but it interfaces with a memory unit (see section 3.7) that handles the low-level transfers with the memory; this decoupling enables to use different kinds of memory very easily.

[5]The maximum number of registers that can be used depends on the number of bits used in the coding of instruction, each register corresponding to a page in the base–page terminology (see section 3.4.2). In the case depicted here, 32 registers can be used per FU.

The instructions are fetched from addresses stored in the program counter (PC) of the fetch unit. This register is automatically updated to point to the next instruction every time one completes.

The PC register also belongs to the inner addressing space and can also be changed from the FUs, a mechanism used for realizing *jumps*. Conditional jumps use the standard mechanism of changing the PC with a special register (the condition register) that indicates, based on its value, if the jump should be taken or not. Different kind of conditions are possible and are computed in another FU, the condition unit (see section 3.8.2).

2. **Instruction decoding**
   Execute directly follows the *Fetch*, with no cycle dedicated to the *Decode* phase. This simplification was made possible thanks to a trivial decoding logic that could be integrated at the beginning of the *Execute phase*. Because of its short execution time – basically, decoding in this case is just a multiplexer – this does not penalize the final working frequency.

   In our processor, we chose to decode the instructions in a centralized way. In that, it differs from the *MOVE32INT* processor in which the decoding has been distributed into the FUs themselves, the instructions being passed directly to every FU.

3. **Transport scheduling**
   When the instructions have been decoded, they are executed by the FUs and, after a clock cycle in the normal case, the next instruction is fetched.

   An exception to this scheme is that, when required, the processor can be put in *Debug* mode, which corresponds to a pause in the execution flow. This can be used for debugging purposes and we used that *halt* mechanism during the development of an *in-circuit debugger* (see section 5.3), which can be used to examine and change all the processor's registers, to execute instructions step-by-step, ... In this mode, instructions are not fetched using the normal scheme and the automatic update of the fetch unit registers (the automatic PC increment, for example) is inhibited.

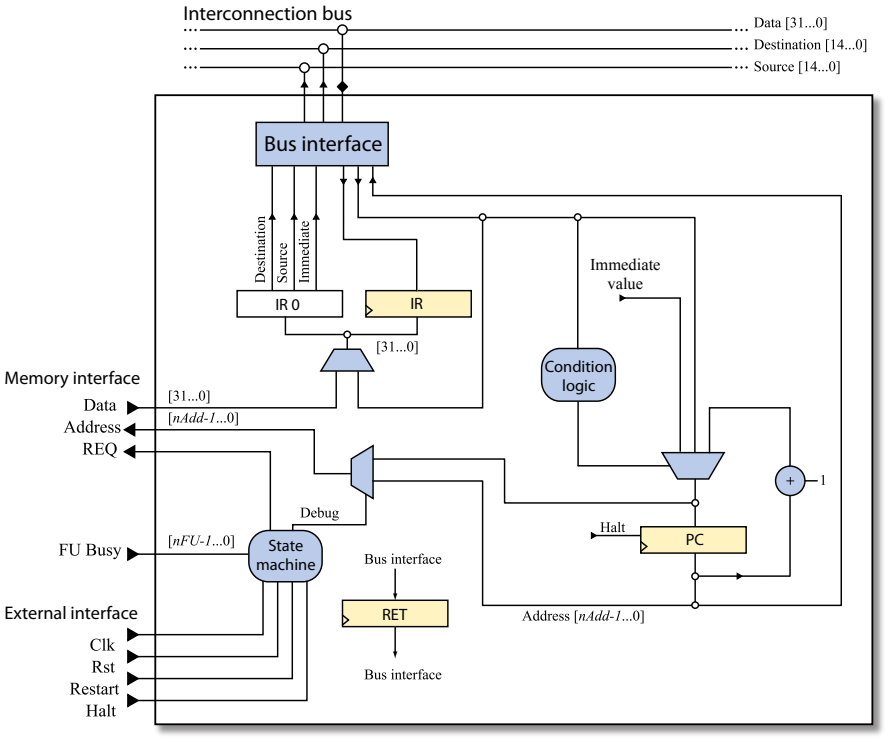


Figure 3.6: The fetch unit.

In the schematic overview of the fetch unit, shown in Figure 3.6, it is worth noting the presence of an instruction register (IR), even in the absence of a dedicated decoding phase[6]. Even if it is not strictly necessary during the normal operation of the processor, the IR is loaded at the same time as the instruction is executed. However, it is not used to assign the source and destination buses which are assigned, like immediate values, directly from the output of the memory unit. Its presence is justified because when the processor is stalled or in debug mode, the memory interface must be released. During that time, buses might need to maintain their value, which is done by using the value stored in the IR. Finally, as the IR is present in the memory map of the processor, it can be used as a source or as a destination like any other register; this permits "exotic" operations to be performed, which can be used for example in the debugger described in section 5.3.

In addition to the PC and IR registers, the fetch unit also contains another register that is accessible in the inner addressing space: the *return address* register (RET). The RET register can be used to store the return address of the caller function during procedure calls. It does not implement any special mechanism to support this feature and, if required, it can be used as a general purpose register. Note that the implementation of this register constitutes only an interesting opportunity offered to the programmer but not an architectural obligation.

3.6.1 Pipelining functional units

Handling operators latencies in TTA is somewhat particular in the sense that the operations are realized between two levels of register that are present at the border of FUs (under the hypothesis that FUs are implementing registers at their addressable locations, which is the case in this work).

When every FU in the processor has an equal latency λ, programming does not require particular attention. Yet, to provide more flexibility in the implementation of FUs, we have chosen to lift this constraint by allowing pipelining inside each FU. Consequently, each FU is free to propose operations that possess *arbitrary fixed latencies* or even *variable latencies*.

Handling different operators latency

Introducing this mechanism yields increasing implementation freedom but presents the disadvantage, from a programmer's perspective, that the latencies of the different operators have to be explicitly handled in assembly, otherwise erroneous values could be used due to a data hazard.

To illustrate the situation, let us imagine an adder FU with a latency $\lambda = 2$ clock cycles. Thus, in the following code sample[7], in order to add two numbers and use the result, the programmer has to insert a nop instruction to wait until the output register is updated with the result of the operation:

```
move add_a, #3;
move add_b_t, #4; // Triggering operation start
nop; // Result not valid yet, wait one cycle
move r0, add_r; // Use the valid result
```

Listing 3.1: Example of code with arbitrary FU latency.

Without the nop, the result obtained would be incorrect because it would not reflect the result of the operation. Thus, the latency of the operator must be taken into account in order not to access the output before it is updated. In theory, any instruction that does not access output register can be used to make the processor wait for the necessary number of cycles. In practice, when no compiler is available, the inserted instructions are often nop for the same reason we mentioned before: the difficulty of mixing different pieces of code in the flow of a program. Because no safeguard is provided when a not yet valid value is used, incorrect programs are frequent.

Another situation that can be problematic arises with operators that have a *variable latency*, i.e. results that may be produced in a different number of clocks cycles. A simple example of such an

[6] In traditional processors, the IR contains the decoded instruction at the end of the *Decode* phase and is used during the *Execution* phase.

[7] Note that the '#' character is used to differentiate an immediate value from an address.

operator is a divider unit: when the divider is equal to 0, the computation does not need to be carried and multiple cycles can be saved if this situation is detected early.

If the first problem can be easily tackled by compiler scheduling, the second problem requires a bit more attention. In [Corporaal 91] and [Corporaal 97], several solutions are examined to deal with FU pipelining:

- *Continue always*. The intermediary stages of the pipeline always update, regardless of the state, possibly producing invalid values.
- *Push/pull*. The intermediary stages are updated only when requested.
- *Hybrid*. This method uses the previous two techniques: values are forwarded to the next stage at each cycle only if they do not overwrite results that have not been used yet.

The last method was chosen in the *MOVE32INT* processor. Its implementation uses a valid bit that permits to determine if a result register is correct or not. If the valid bit is not set when the socket is read, the processor halts and waits.

The busy bit mechanism

ULYSSE uses a technique that enables the FUs to implement any suitable technique at the FU level directly. This is a very flexible solution in which each FU is free to use any scheme it might find useful, a possibility that also includes mimicking the behavior of hybrid pipelining.

This general behavior is achieved with the definition of a *busy* signal that each FU has to implement. Thus, an instruction requiring more than one cycle is triggered like any other instruction but if the result is read or another trigger is tried whilst the unit has not yet finished (i.e. its busy signal is asserted), the fetch unit halts until the unit becomes accessible again.

In any case, execution cycles are lost *only* in situations where invalid results would have been generated, which combines two advantages. First, it is not necessary to explicitly take into account the latencies of the operators when programming in assembly. Second, the possibility to take advantage of smart scheduling with a compiler is preserved. In addition, this mechanism enables operators that implement *variable latencies*, a situation that even a compiler could not solve without implementing worst case scenarios (schedule the usage of a result always after the worst case of the variable delay). In other words, the programmer does not need to know anything about latencies expect unless he or she is interested in taking them into account.

In a sense, the *busy bit* mechanism simply moves the problem of the latency from software to hardware, which of course translates into a more complex decoding logic that requires to examine whether an accessed unit is busy or not before dispatching the next instruction.

One can also wonder if this additional cost, especially in the TTA approach that aims at simplifying hardware as much as possible, is justified. Considering the fact that in the absence of a compiler, this mechanism acts as a safeguard and helps saving time and that when a compiler is used, it still enables the use of variable latencies, we think the answer is positive.

Busy bit implementation

To take into account variable latencies, each FU must then provide a *busy* bit to the fetch unit (the bit can be forced to 0 if the FU is using no pipelining mechanism). This bit indicates whether the unit is ready to receive new operands or if its output is valid. The fetch unit is thus informed, when it decodes instructions, if the addressed units are ready as source or destination. If it is not the case, the fetch unit waits until the busy flags revert to their "ready" state before continuing.

In the fetch unit source code (Listing A.3, annex p. 212), it can be seen that even if the principle of the mechanism is simple, its implementation requires several corner cases to be considered, which augments considerably the complexity of the component.

FU pipelining and scheduling opportunities

Overall, the *busy bit* and hybrid pipelining both present the advantage of high scheduling freedom, because they allow to decouple the different kind of moves that can occur in the processor: *move to an operand socket*, *move to a triggering socket* and *move from a result socket*. Thanks to this separation, different scheduling advantages are then possible:

- Because operands can be updated before the triggers happen, the result of the FU remains valid and can be used until a new trigger is activated.
- A move from the result register can happen before, when, or after λ cycles have passed since the triggering happened. If it happens before, the CPU will halt, and if it happens after, the operation is handled normally because the result is still available in the pipelined unit. As pointed out by Corporaal in [Corporaal 97, p.137]: "*This way results may stay longer in the FU pipeline; this enhances the direct bypassing of results to other FUs* [...]".

3.7 Memory interface unit

The memory interface is used to connect different memory types with the processor. Two different controllers were developed, one for a fully embedded version of the processor that does not require external memory and an other containing an asynchronous SRAM controller. Both share the same access methodology but propose different performance and size requirements.

3.7.1 Embedded memory controller

The embedded controller is relatively simple and is targeted for Xilinx FPGAs that contain *SelectRAM* memory blocks[8]. Because these memory blocks are dual-ported and have a one-cycle latency, using this controller enables the processor to work at maximum frequency with maximum instruction throughput (up to one instruction per cycle). Yet, this comes at the cost of a memory size limited to the *SelectRAM* blocks available in the FPGA, an amount that depends on the exact FPGA model but that remains relatively low (in the range of a few thousand 32-bits words).

Address	*RW mode*	*Trigger*	*Name – Function*	*Description*
0	RW	No	Mode	Access mode of the `DP` register
1	RW	No	DP	Data pointer register
2	RW	Yes	Data	Read or write a data

Table 3.1: Embedded memory controller memory map.

As shown in Table 3.1, memory access is straightforward with this controller, `load` and `store` operations corresponding to reading or writing the *Data* register. Changing the accessed address for reads and writes is done by modifying a memory pointer called *DP*. The interface itself proposes three different memory access types that can be used to simplify operations such as array accesses. These different modes can be chosen with a configuration register (named `MODE`) that can modify automatically the `DP` register, hence requiring fewer operations to access various memory locations:

1. *Direct access* - DREAD mnemonic
   This is the standard way of accessing memory, the `DP` pointer is not modified automatically.

[8]These are dual-port embedded memory blocks that are present on many Xilinx FPGAs. See `http://www.xilinx.com/support/documentation/topicmeminterfacestorelement_ram-rom.htm`

2. *Auto-incremented mode* - AUTOINC mnemonic
   Every read or write operation in memory is followed by an automatic increment of the `DP` pointer. This mode can be used to read consecutive elements in an array.

3. *Stack mode* - STACK mnemonic
   After each read operation, the `DP` pointer is decremented (operation `pop`) and every write operation triggers the increment of the `DP` (operation `push`). With this mode a stack segment can be easily generated, which is useful for implementing function calls, for example. The stack realized is of "drum" type, which means that when one of the boundaries of the memory is reached, the memory access is wrapped to the other end. For instance, if the pointer equals to 0 and a `pop` operation is realized, the pointer will be updated with the highest address. Note that the position of the pointer is updated after the operation itself (post auto-modified scheme).

3.7.2 Asynchronous SRAM FU and controller

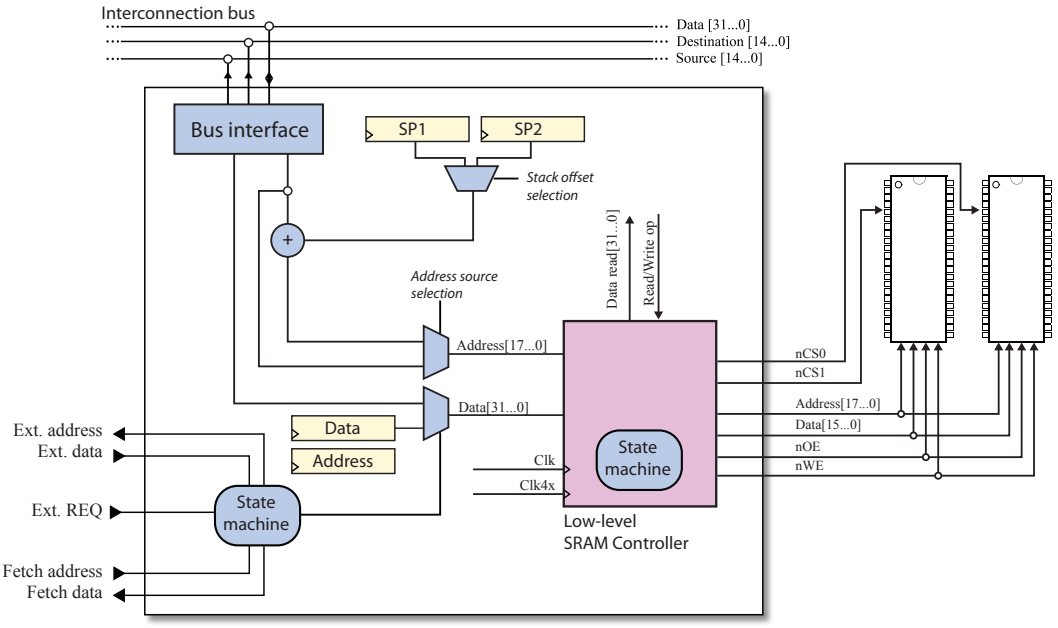


Figure 3.7: The SRAM controller FU schematic.

This memory controller (Figure 3.7) is more complex and enables access to asynchronous SRAM memory chips. It provides three main functionalities:

1. It provides instructions to the fetch unit. Read requests issued by the fetch unit are handled by a dedicated bus between the fetch unit and the controller.

2. It provides access to the memory from the program application: because no dedicated instructions exist to access memory in the architecture, `load` and `store` operations are "emulated" by the reading and writing of an address value at particular locations. These accesses modify a register called `DATA` as follows: when a `load` operation occurs, the resulting value can be read in this register; a `store` operation requires to write the value to be stored into memory to this register first and then trigger the store operation by writing the address to the corresponding trigger register.

   Two registers, `SP1` and `SP2` can be used to automatically compute addresses with offset values, which can be handy and save some execution cycles for example in function calls.

3. Loading instructions at startup. When started, the CPU is automatically put in the *Debug* state because the memory content is invalid. To update the code of the CPU, every version of the processor contains a mechanism that has access to external pins so that the code can be loaded (using different protocols) at startup. In the *standalone* revision of the processor, a serial loader is used whereas in the *cellular* revision of the processor, code is loaded via the on-chip network (more on this in chapter 7).

   In both cases, a particular FU can gain access to the SRAM controller by asserting the appropriate signals (named `EXT_REQ` in Figure 3.7) to indicate that it requests access to the memory chip controller. During that time, the fetch unit can not access the memory anymore and must wait. All these implementation details can be found in the annex, from page 212 onwards.

Due to the nature of the SRAM memory modules and because of some design constraints that will be described below, read and write operations can not be performed in one clock cycle. This imposes some slight modifications to the fetch unit, as instructions can not be read in one cycle and also because load and store operation would interfere with instruction fetching (we are using an Von Neumann memory model, where instructions and data are mixed in memory and use the same storage structure). As a result, the state machine is changed as shown in Figure 3.8.

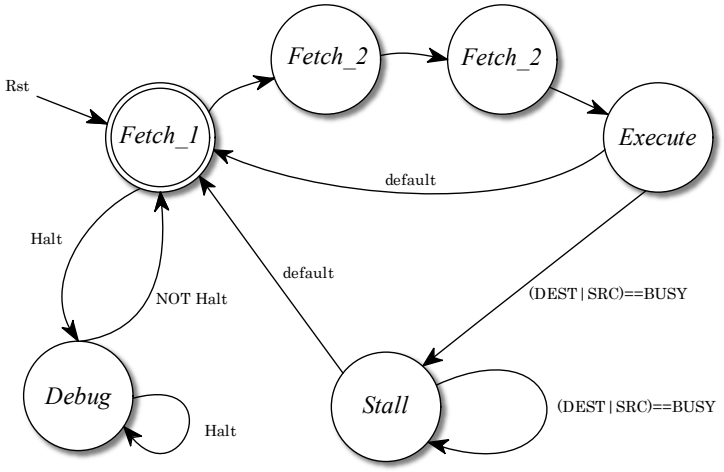


Figure 3.8: Fetch unit state-machine when using external SRAM memory.

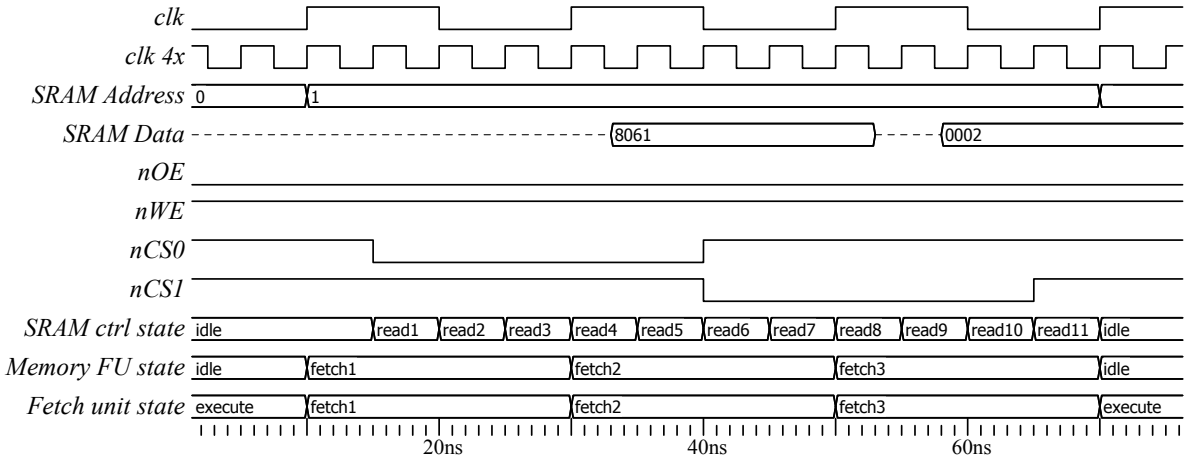
Hardware platforms note

One of the reasons why one-cycle operations are not possible is that in every hardware platform used in this project that possesses external memory (i.e. the *standalone* and *cell* versions), the processor interfaces with two Samsung 10ns 16-bit wide SRAM memory chips[9]. Due to their asynchronous nature, memory timings play an essential role in these components because timing violations lead to corrupted memory data. In addition, due to the limited number of pins of the target FPGA, the data lines between the two chips are *shared*, which translates into the fact that reading or writing 32-bit values requires two memory accesses (using two different *chip-select* signals `CS0` and `CS1`).

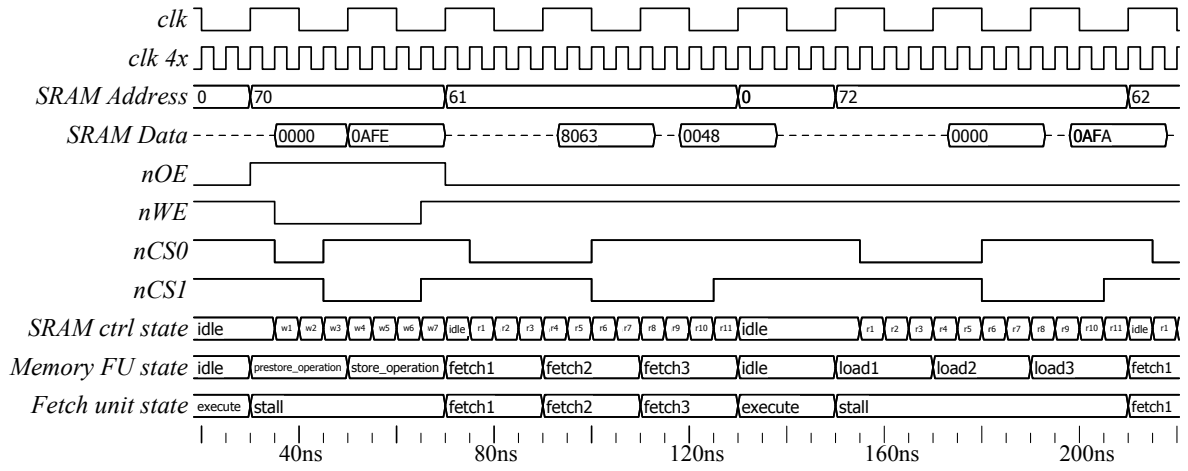
Micro-cycles, simulation and realization

The other reason why one cycle operations are not possible is that the controller was realized using synchronous logic and the memory chips are asynchronous. As a result, several problems appeared and precautions had to be taken to ensure correct behavior. Standard IP controllers (for instance on

[9]Exact model designation is Samsung K6R4016V1D.



(a) Fetch and execute



(b) Effects of a load after store

Figure 3.9: SRAM controller timings.

OpenCores[10]) normally require four clock cycles per read and write operation. However, using such a component would have required eight clock cycles per memory read, which would have hindered performance too much. To avoid this situation, we decided to optimize access as much as possible by implementing our own controller.

Namely, we developed a controller using a separate clock running four times faster than the CPU clock, which was used to create micro-cycles inside the normal cycle period. Thus, using a clock frequency of 50 MHz as an input, the fast clock generates four 5ns long cycles for every cycle. The fast clock drives a state machine that is used to ensure that all setup and hold constraints of the memory are met in every possible case.

We had to use a very precise timing simulation model of the memory chips (with adequate setup and hold time checks) to ensure correct behavior. In addition, an unexpected problem that arose during the development of this component was that the different timing constraints also had to be still valid *outside* the FPGA so that the memory chips could work. In fact, because of the routing and placement delays inside the FPGA, signals generated during the same clock edge could present a few nano-seconds of difference when measured at the FPGA pins.

To ensure correct behavior even in this case, synthesis constraints had to be added. To validate the

[10] `http://www.opencores.org`

model on the actual hardware, the complete simulation model also includes place and route routing delays to reflect as much as possible the real situation.

Figure 3.9 shows both a standard instruction fetch (3.9a) taking 3 cycles and a more interesting waveform (3.9b) in which a load operation (3 cycles) is performed after a write operation (2 cycles), showing all the accesses of the controller. One can note that because memory *load and store* operations take more than one cycle, the fetch unit should be prevented from reading the next instruction while a memory access operation is in progress. This is done using the busy bit mechanism.

Memory map

Address	*RW mode*	*Trigger*	*Name – Function*	*Description*
0	RW	No	SP1	Stack pointer number 1
1	RW	No	SP2	Stack pointer number 2
2	RW	No	Data	Memory data register
3	W	Yes	Load absolute	Data ← Memory(Address)
4	W	Yes	Store absolute	Memory(Address) ← Data
5	W	Yes	Load with offset 1	Data ← Mem(Address + SP1)
6	W	Yes	Load with offset 2	Data ← Mem(Address + SP2)
7	W	Yes	Store with offset 1	Mem(Address + SP1) ← Data
8	W	Yes	Store with offset 2	Mem(Address + SP2) ← Data

Table 3.2: SRAM FU Interface.

To close the discussion regarding this functional unit, we present in Table 3.2 a high-level perspective of the functional unit showing how the unit is seen from a programmer's perspective.

3.8 ULYSSE functional units

As shown in Figure 3.10, several functional units were developed as part of this work to accommodate the various needs of the different versions of the ULYSSE processor. We will not detail them here but the annex contains (see section A.2, p. 206) a description of the interface of these various operators as well as more practical information related to their operation. They can be classified into four main categories (see Table 3.3):

1. **Arithmetic**
   The first category contains *arithmetic operators*, i.e. FUs used to compute mathematical and logical functions.

2. **Interface**
   The second category contains units used to *interface* ULYSSE with the outside world.

3. **General**
   The third category is composed of units used in *general operation* of the ULYSSE processor or as internal peripherals.

4. **Debug**
   The last category contains functional units that are used for *debugging* purposes, by providing simulation or debugging mechanisms.

In the remaining of this section, we will briefly focus on some operators that represent a stretch to the TTA philosophy or that present some interest from a theoretical standpoint.

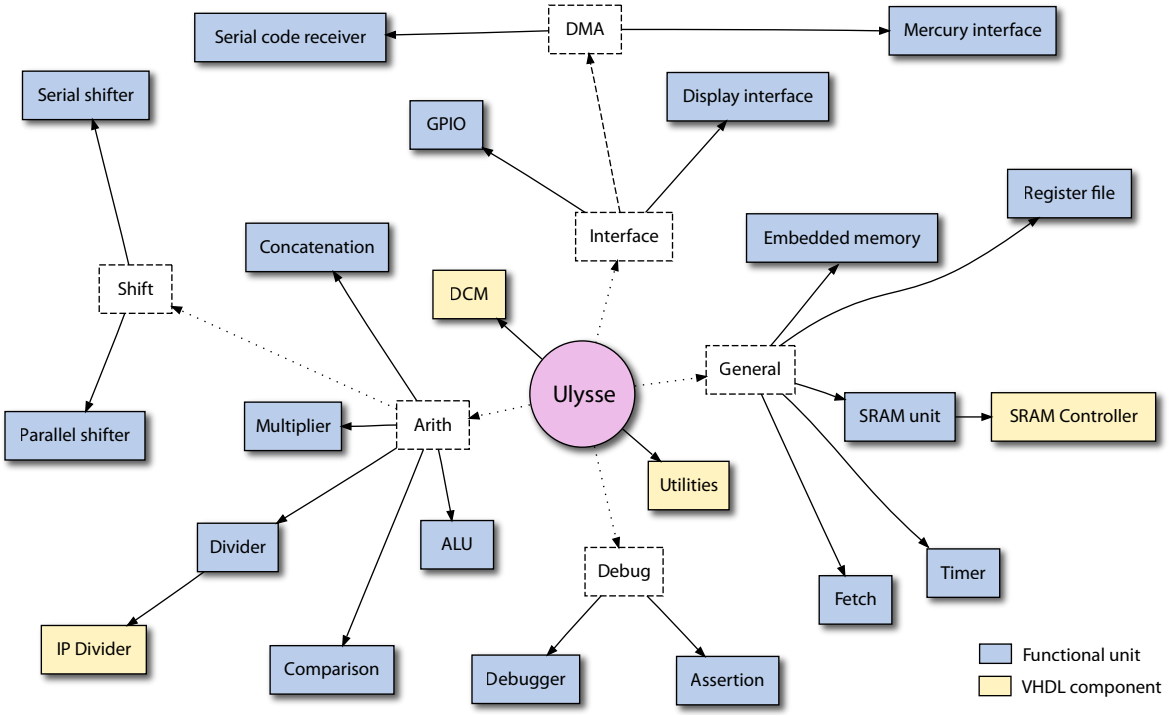


Figure 3.10: ULYSSE functional units hierarchy.

Name	*Latency* λ	*Description*
ALU	2	Common arithmetic/logic operations: $+$, $-$, $\vee$, $\wedge$, $\oplus$, $\neg$, $\times$. See 3.8.3
Shift sequential	Var[a]	Sequential version of `SHL`, `SHR`, `SHRA`, `SHR`, `ROTR`, `ROTL`
Shift parallel	2	Parallel version of `SHL`, `SHR`, `SHRA`, `SHR`, (`ROTR`, `ROTL` optional)
Concatenation	2	To concatenate numbers. See 3.8.1
Divider	43	To compute a/b and $a \bmod b$
Comparison	2	Allows to do various comparisons between two numbers. Also determines if a number is positive, negative or null. See 3.8.2
GPIO	2	Simple FU to access the external input and output pins of the circuit
Code receiver	N/A	Can be used to access the SRAM memory of the processor from the outside. Used to upload the code when the processor is used in *standalone* mode
Mercury unit	N/A	Used in the *cellular* processor configuration as a high-speed network interface to a routing system. It will be detailed in section 8.4.1
Display interface	2	Used in the *cellular* processor configuration to interface with a LED display. It will be detailed in section 8.4.2
Register file	2	Contains up to 32 general purpose registers
Timer	2	A resettable timer that provides microsecond resolution to measure time
Fetch unit	N/A	Fetches instructions and executes them. See section 3.6
SRAM interf.	2	An interface to asynchronous SRAM memory chips. Provides `LOAD` and `STORE` operations. See 3.7
Embedded interf.	2	An interface to the Xilinx SelectRAM dual-ported memory blocks for `LOAD` and `STORE` operations. Used only in the VLIW version of the processor
Assertion unit	2	Generates simulation exceptions and debug messages. See 5.1
In-circuit debug.	N/A	In conjunction with the fetch unit, allows to pause and resume processor execution. See section 5.3

[a] Depends on the number of shift operation performed.

Table 3.3: Arithmetic FUs available for the ULYSSE processor.

3.8.1 Concatenation unit

As described in the instruction encoding (section 3.4.2), immediate values are stored in the instruction as 16-bit values.

If a bigger range is required, it is of course possible to compute bigger values with masks and shift operations. However, since 32-bits immediate values can be relatively common (and also because ULYSSE is a full-fledged 32 bit processor), we implemented an unit whose purpose is to realize the concatenation of values.

The unit is really simple as it contains only two inputs (A and B) and one output (R). It realizes the following mathematical operation, which is trivial to implement in hardware (consisting of only wires): $R = (A \ll 16) \vee B$.

3.8.2 Conditional instructions (condition unit)

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	OpA	Operand A
1	RW	No	OpB	Operand B
2	RW	No	OPcode	Selects the operation performed
3	R	No	Result	Depending on OPCode: $A < 0$, $A > 0$, $A = 0$, $A \neq 0$, $A > B$, $A \geq B$, $A < B$, $A \leq B$, $A = B$, $A \neq B$

Table 3.4: Condition functional unit.

The interest of this functional unit resides in the fact it is normally used to compute conditions for `jump` operations. In fact, the fetch unit can generate two different kinds of jump operations: either *immediate* or *conditional immediate*. In the first case, the jump is always taken and in the second it is taken if and only if the condition register of the fetch unit is equal to 1.

Even if the condition register can be updated by any FU, the condition unit provides several methods to "emulate" standard jump conditions, as summarized in its memory map, shown in Table 3.4. It can be remarked that we are using a different mapping mechanism for this operator in the sense we are using an *OPcode* register that selects among the different operations realized, which all have a latency $\lambda = 2$.

This mechanism was chosen due to the large number of operations this FU can perform, which is in fact larger than the number of registers per FU that the VLIW version can address[11].

3.8.3 Arithmetic and logic functional (ALU) unit

As we mentioned earlier, most ULYSSE implementations do not implement parallel displacements in the form of VLIW instructions. Because of this, it becomes possible to regroup multiple arithmetic and logic units together since any scheduling advantage offered by separated FU is lost. Moreover, this grouping also allows to reduce the cost of multiple bus interfaces, which is positive even if bus interfaces in themselves are not so costly in terms of hardware.

The ALU FU, for example, regroups many of the mathematical operations that are normally used in a processor. Its memory map is shown in Table 3.5. Every operation has a latency of $\lambda = 2$ and the functional unit also presents the advantage that every one of its operations is done in parallel when the functional unit is triggered. The results of the different operations can then be read at different locations. This means that only one displacement is required to do every possible ALU operation, which in turn enables interesting scheduling opportunities because the results of all these operations can be retrieved directly.

[11] Because the internal addressing space in this version is smaller than the others, only 3 bits are devoted to *page addresses*, which restricts the maximum number of registers per FU to 8.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	No	OpA	Operand A
1	RW	Yes	OpB	Operand B
2	R	No	Add	$A + B$
3	R	No	Substract	$A - B$
4	R	No	`OR`	$A \vee B$
5	R	No	`AND`	$A \wedge B$
6	R	No	`NOT`	$\neg B$
7	R	No	`XOR`	$A \oplus B$
8	R	No	Multiply$_{low}$	$(A \cdot B)_{[31\ldots0]}$
9	R	No	Multiply$_{high}$	$(A \cdot B)_{[63\ldots32]}$

Table 3.5: ALU functional unit.

3.9 Hardware performance evaluation

In this section, we will examine the performance of the ULYSSE processor from a hardware perspective, i.e. focusing on hardware requirements such as size estimations per FU, achievable frequencies.... This analysis will be completed, in chapter 5 (section 5.5), after programming tools such as the compiler will be introduced, by an extensive analysis of the processor including different software benchmarks.

Our discussion will be targeted on the *cellular* version of the processor because it is the most complete and the most interesting in the context of this thesis.

3.9.1 Ulysse's FU size and speed

Table 3.6 summarizes the hardware related information computed using the *Xilinx ISE 10.1* tool suite along with the *Synplify Pro 9.0.1* logic synthesizer. The functional unit figures are estimates based on the synthesis of the functional unit with its complete bus interface. The three last rows of the table represent final CPU size and speed results based on a post-place-and-route analysis.

3.10 Possible further developments

Developing a processor from the ground up takes time and can be considered as an endless task, new operators and increased performance being constant objectives. In its current state, the processor could benefit from the implementation of *caching* techniques so that it could reduce its CPI, notably when using external memory. Of course, more performance could also be attained by overcoming the weaknesses related to the VLIW revision. Finally, designing a FU to access (or bridge) industry-standard buses such as ARM, AMBA [Flynn 97], or the IBM CoreConnect series (OPB/PLB)[12] would also allow various existing IP modules to be used, be they free or commercial.

Besides implementation-related improvements, one research opportunity that could be interesting to follow would be to push the idea of *pluggable FUs* further by extending the architecture so that it could support *dynamic reconfigurability*. Consisting of changing the configuration of parts of the reconfigurable hardware on line, this technique could enable to use free zones that could implement new operators suitable for different algorithms depending on the executed program. While similar approaches have been proposed in the past (for example [Epalza 04, Jain 04]), the *Move* paradigm proposes useful features that other architectures lack.

[12] `http://www.xilinx.com/support/documentation/ip_documentation/opb_v20.pdf`

Functional unit description	*LUT*	*Register bits*	*Max freq.*
ALU without 32 × 32 multiplier	493	418	> 200 MHz
ALU with 32 × 32 multiplier[a]	525	496	> 200 MHz
Parallel shifter, without `rot` ops	606	245	> 200 MHz
Parallel shifter, with `rot` ops	1005	486	> 200 MHz
Concatenation	76	32	> 200 MHz
RF (8 registers)	256	400	> 200 MHz
SRAM memory	399	323	> 200 MHz
Condition unit[b]	254	272	172 MHz
Timer	59	40	180 MHz
Fetch unit	252	121	132 MHz
Code uploader	80	61	140 MHz
Divider[c]	> 452	> 762	> 50 MHz
Routing interface[d]	111	85	100 MHz
Mercury receiver	299	190	157 MHz
Smallest[e] CPU	756	362	52 MHz
Usable[f] CPU	2562	1154	54 MHz
Full CPU (after PR)	2975	1371	53 MHz

[a] Uses 4 multiplier blocks
[b] Critical path on the "=" operation
[c] Could only provide lower bounds because IP module can not be fully estimated due to license restrictions
[d] Uses 1 BRAM module
[e] Support FUs only
[f] 69% of Spartan3–XCS200 FPGA; no mult, no timer, no cat unit

Table 3.6: Size and speed of available ULYSSE functional units.

3.10.1 Permanent connections and meta-operators

We propose here a method that enables, by adding a certain number of buses, to increase *ILP* so that more instructions are executed than instruction *slots* are present. Even if this technique was not implemented due to time restrictions, we think its description could lead to an interesting increase of performance.

In fact, as we have already mentioned, a possible method to increase the performance of a processor is to increase ILP, something TTA enables thanks to VLIW instructions. However, this parallelism is limited by the number of slots per instruction. In case of repetitive operations, it might be interesting to specify that a displacement must be *permanently* done (at each instruction) until further notice.

As the operators implemented possess for the most part a relatively low latency and because it is simple to move data in series and in parallel, it might be interesting to introduce the concept of *meta-operator*.

To illustrate the concept, let us consider a standard DSP instruction such as the `MACC` (*multiply-and-accumulate*). Let us now imagine that this operation is not directly present as an FU but that we want to use it anyway: with one *multiply* and one *add* FU, the code corresponding to the `MACC` operation is the following:

```
/* MACC operation equivalent*/
loop:
      move multiplier_a_t, operand1;
      move multiplier_b, operand2;
      move adder_a_t, adder_result;
      move adder_b, multiplier_result;
      [...]
```

As the accumulation of results is done during the whole loop, the second displacement could be specified as *permanent* and done in parallel, reducing the size of the operation kernel. Figure 3.11 represents the required configuration of an automatic `MACC` operator, which corresponds to the

following code:

```
/* MACC operation with permanent connections*/
move_perm adder_a_t, adder_result;
move adder_b, multiplier_result;

loop:
      move multiplier_a_t, operand1;
      move multiplier_b, operand2;
      [...]
```

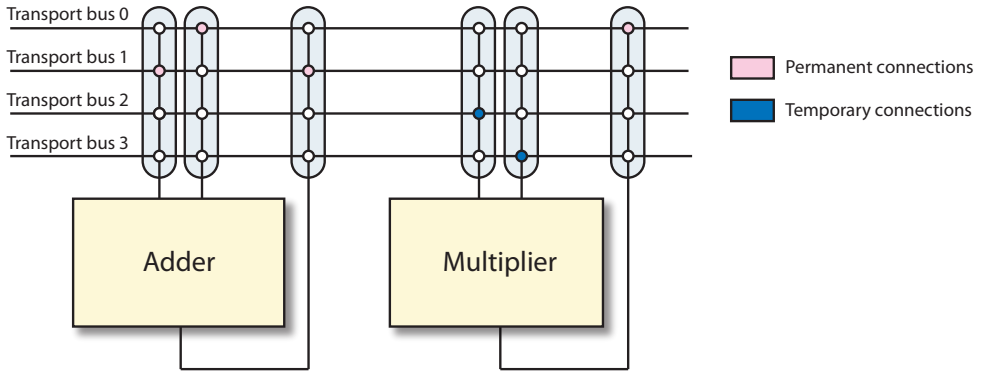


Figure 3.11: MACC with permanent connections.

The concatenation of the multiply and add operations thanks to permanent connections is what we call a *meta-operator*. Its latency is equal to the sum of the latencies of the pipelined functional units. In the general case:

$$\lambda_{tot} = \sum_{i=0}^{n} \lambda_i \qquad n \text{ being the number of operators in series} \tag{3.3}$$

Which yields

$$\lambda_{tot} \in \mathcal{O}(n) \tag{3.4}$$

The working frequency of the resulting system will depend in the worst case of the slowest functional unit. As in every operator the critical path traversal time δ is short because it lies between two registers, the resulting pipeline can also work at a high frequency. More formally, we have that:

$$\forall i, i \neq max \Rightarrow \delta_{max} > \delta_i \qquad max \text{ being the latency of the slowest operator} \tag{3.5}$$

$$\delta_{tot} = \delta_{max} \Rightarrow \delta_{tot} \in \mathcal{O}(1) \tag{3.6}$$

Of course, the use of permanent connections requires supplementary communication buses. Furthermore, the assembler or the compiler must be able to take these permanent connections into account to determine the existence of free transfer buses. In counterpart, processor parallelism is increased as more displacements than *slots* can be realized.

Research in this direction could also try to examine automatic allocation and assignation of buses, notably by analyzing if an algorithm could determine an optimal transfer scheme, at the compiler level for example, between permanent or temporary moves.

3.11 Conclusion

With the description of the ULYSSE processor from an hardware standpoint, we were able to demonstrate that the architecture suits our needs by providing various levels of flexibility, such as the *pluggability* of FUs or the pipelining opportunities with the *busy bit* mechanism, while remaining relatively cost effective.

One of the main goals of the implementation was to validate the model and to show that it was able to provide the user with enough flexibility to fulfill its task as a bio-inspired processor. Adding a new operator is simple and straightforward: by implementing the VHDL bus interface and specifying the number of accessible registers in the FU, the component is automatically added to the memory map of the processor. Of course, this must be done *before* synthesis. Still, it is possible to define an *ad-hoc* processor that can be specific for each application very easily.

The following example shows how this translates in the entity of the processor itself:

```
entity ulysse_cell is
  generic
  (
    clk_frequency      : positive := 50000000;
    alu_present        : positive := 1;
    enable_multiplier : boolean := true;
    concat_present     : positive := 1;
    rf_present         : positive := 1;
    divider_present    : positive := 0;
    shift_present      : positive := 1;
    enable_rot_op      : boolean := false;
    has_timer_1        : boolean := true;
    has_timer_2        : boolean := true;
    mercury_interface_present : boolean := true;
    routing_interface_present : boolean := true;
  );
```

With this interface, changing the presence or even the number of instances of a particular FU in an implementation is only a matter of changing the generic parameters of the processor.

In the next chapter, we will examine how the achieved hardware flexibility translates at the software level and also how it can be leveraged to ease development of programs and also of the tools themselves.

Bibliography

[Corporaal 91] Henk Corporaal & Hans (J.M.) Mulder. *MOVE: a framework for high-performance processor design*. In Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 692–701, New York, USA, 1991. ACM.

[Corporaal 93] Henk Corporaal & Paul van der Arend. *MOVE32INT, a sea of gates realization of a high performance transport triggered architecture*. Microprocessing and Microprogramming, vol. 38, pages 53–60, 1993.

[Corporaal 97] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, USA, 1997.

[Epalza 04] Marc Epalza, Paolo Ienne & Daniel Mlynek. *Dynamic reallocation of functional units in superscalar processors*. In Proceedings of the 9th Asia-Pacific Computer Systems Architecture Conference, pages 185–198, Beijing, August 2004.

[Flynn 97] D. Flynn. *AMBA: enabling reusable on-chip designs*. IEEE Micro, vol. 17, no. 4, pages 20–27, Jul/Aug 1997.

[Hoffmann 04] Ralph Hoffmann. *Processeur à haut degré de parallélisme basé sur des composantes sérielles*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[Hoogerbrugge 96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[Jain 04] Diviya Jain, Anshul Kumar, Laura Pozzi & Paolo Ienne. *Automatically customising VLIW architectures with coarse grained application-specific functional units*. In Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04), pages 17–32, Amsterdam, September 2004.

[Lin 04] C.H. Lin, Y. Xie & W. Wolf. *LZW-based Code Compression for VLIW Embedded Systems*. In Proceedings of the conference on Design, automation and test in Europe (DATE'04), pages 76–81, 2004.

[Myers 81] Glenford J. Myers. *Advances in Computer Architectures*. Wiley-Interscience, 1981.

[Xie 01] Y. Xie, W. Wolf & H. Lekatsas. *Code Compression for VLIW Processors*. In Proceedings of the 34th International Symposium on Microarchitecture (MICRO 01), pages 66–75, 2001.

Chapter 4

Development tools

"Well, less is more, Lucrezia."

ROBERT BROWNING, *Andrea del Sarto*

ONE OF THE dangers in designing complex hardware systems in an academic setting is that the complexity of the development, coupled with the difficulty to adequately test the hardware, often monopolizes all the resources, to the expense of software support. TTA processors do escape this rule, but their modularity enable the application of interesting techniques for their testing. In counterpart, their programming in assembly language is more complex than traditional processors because they do not directly provide the programmer the traditional abstractions he or she expects.

The subject of this chapter is to analyze how the latter issue can be tackled with different techniques such as *macros* and how modularity can be exploited to simplify the testing of the processors. This discussion will focus on the ULYSSE processor, but the underlying principles could be, in general, applied to any *Move* processor.

The second part of this chapter will focus on the peculiarities of *Move* programming and how they influenced the development of an assembler for the processor. In section 4.2, we will analyze some of the difficulties encountered during the port of the GCC compiler for our processor.

4.1 The assembler

The ULYSSE assembler is a software tool that transforms assembly code into its executable binary form. In this section, we will show that the use of a single instruction, because of the narrow perspective offered, limits the low-level programmability of such a processor. To overcome this limit, we will develop the notion of *macros*, an abstraction that offers a higher-level view of the processor while conserving its basic operation.

Because it also constitutes an essential component of the software tool-chain for the ULYSSE processor, the assembler has to be carefully designed and a choice must be made among the different implementation options offered. In our case, we preferred to focus on the back-end of the tool (the code generation) by using an existing *Java* grammar parser generator (explained in section 4.1.9) that considerably simplifies the use of multiple passes during the assembly.

4.1.1 Assembly process

The different assembly phases (shown in Figure 4.1) through which the assembly source code must be processed are the following:

1. **Lexical verification**
   During this first phase, the part of the assembler called the lexer makes sure that the input program is compatible with the language grammar. The comments are eliminated along with the

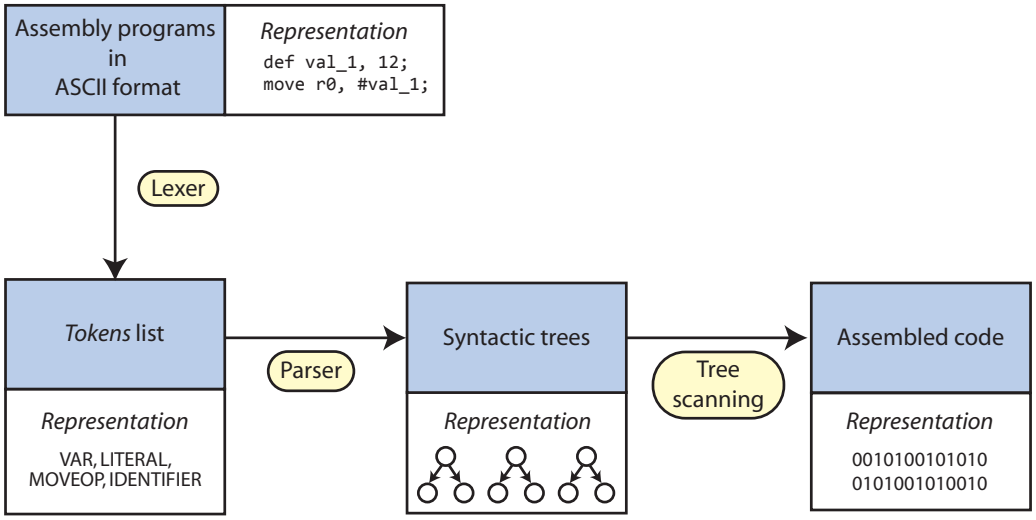


Figure 4.1: Description of the various phases of assembly.

carriage returns, separators such as ";", etc. After this phase, the program has been transformed in a set of *tokens*. This pass is also used to take care of variables declaration: each time a variable is encountered, it is stored in a *hash table* (the key being the name of the variable) that will be used later on to verify, for example, if a variable has been already declared.

2. **Syntactic verification**
   During this phase, the assembler verifies that the program follows the language syntax and produces the syntactic tree corresponding to the program. Depending on whether an instruction contains an immediate value or not, two nodes are possible:

   (a) *Displacement address, immediate*
   The left child is the destination address and the right child is the immediate value. The destination address and the immediate values are stored in the children nodes.

   (b) *Displacement address, address*
   The left child is the destination address and the right child is the source address. The addresses are stored in their respective nodes.

   This parsing phase was implemented using the *Visitor design pattern* which was defined by Gamma et al. in their seminal work on design patterns [Gamma 95], which presents programming practices that can be applied to several common programming problem. We also refer the interested reader to [Palsberg 98], which gives a more specific overview of the *Visitor* pattern.

3. **Code generation**
   This last phase generates the executable code of the processor. The generation is done by recursively parsing the nodes of the syntactic tree produced during the second phase. Label addresses are computed and variable declarations are checked.

A simple example

We will illustrate now how the phases of the assembly process are applied to a simple program:

```
def r0, (9, 0); // Declaration of the R0 register using its paginated address
def v1, 25; // Declaration of an alias for the constant value 25
move r0, #v1; // Displacement of the immediate value 25 to the register r0
```

After the lexical analysis, the transformation info *tokens* are shown in Table 4.1.

`def`	↦	VALDECL
`r0`	↦	IDENTIFIER
`(9,0)`	↦	LITERAL
`move`	↦	MOVEOP
`#v1`	↦	IDENTIFIER
`25`	↦	LITERAL

Table 4.1: *Tokenization* of the input file.

As can be noted, the typographic decorations as well as the separators are absent. The syntactic analysis then produces the tree depicted in Figure 4.2.

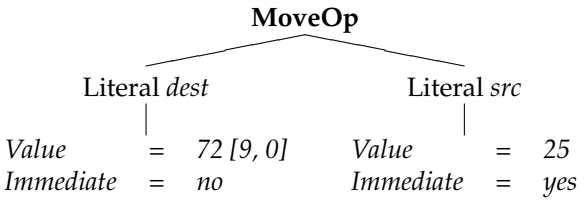


Figure 4.2: Decorated syntactic tree

During the last phase, the tree of the program is parsed using the `Visit()` method of each node, which corresponds during the code generation phase[1] to the generation of the necessary transports in the output file, using the instruction format defined in section 3.4.2.

4.1.2 Memory map definition with file inclusion

Realizing the assembler of a TTA processor implies that the ALU operations that are present in a standard RISC processor must be represented as addresses in the memory space of the processor[2]. As a result, a TTA processor possesses a different relation with its assembler than other processors. In fact, a program assembled for a specific CPU embedding a given set of functional units can also be compatible with a CPU using another set of FUs, as long as they include the operation set of the first CPU and that the common FUs are located at the same addresses in the memory map.

Code compatibility criteria

More formally, let us now define an operation set $OP = \{op_1, op_2, \ldots, op_n\}$. In addition, let us define a processor CPU_1 with an operation set A with $A \in OP$ and a processor CPU_2 with an operation set B with $B \in OP$.

If CPU_1 runs a program P that uses every one of its operations, the same program can be run without reassembly on a processor CPU_2 if and only if the two following conditions are respected (the *code compatibility criteria*):

$$1) \quad A \subseteq B \tag{4.1}$$

$$2) \quad \forall op_i \in A, address(op_i, CPU_1) = address(op_i, CPU_2) \tag{4.2}$$

Where $address(x, y)$ is the address of the operation x in the memory map of the processor y.

To be able to exploit the potential of such a property, we decided to split the processor FU memory map definition from the programs themselves so that they could be assembled for different processor

[1] More details can be retrieved in the comments of the assembler source code.

[2] Note that the assembler can use both linear and paginated representations indifferently.

versions without having to change the code itself. The code related to the execution of instructions is thus clearly separated from the hardware dependent source code.

4.1.3 Definition of new operators

The memory mapping mechanism is also interesting when new operators in the form of FUs have to be added, because assembler support is direct: the operator memory map simply has to be added to the memory map definition file and it can be then directly used. Modifying in a similar way an assembler for a RISC processor, for example, would require the assembler code to be modified and then recompiled, at least when changing basic operations such as `add`. In TTA, as everything can be considered as a data displacement, even basic operations can be easily and rapidly changed.

One development that could be interesting to harvest the flexibility of the system would be to automate the generation of the memory map that corresponds to the program to be executed. By determining the set of operators required to execute a given program, a *minimal* custom processor could be automatically generated along with its memory map (see notably on this subject [Hoffmann 04, chapter 19]). Thus, different processors would be easily generated for different programs, with the advantage that only useful FUs would be present.

4.1.4 Macros and their influence on the definition of a meta-language

Because a n-operand, m result operation can be translated into $n+m$ `move` operations [Corporaal 99], writing programs using only data transports is a very complex task, at least when realized by hand at the assembly level. The very low level vision renders basic operations fastidious. Let us now consider a small code snippet:

```
while(c != 0){
      c--;
}
```

A direct translation to non-optimized *Move* assembly, if we suppose variable c is stored in register `r0`, gives:

```
loop:
      // Compute c−1 and put the result in r0
      move adder_a, r0;
      move adder_b_t, #-1;
      move r0, adder_r;

      // Check if r0 == 0
      move cond_op, #zero;
      move cond_a_t, r0;
      move jump_condition, cond_result;

      // If r0 != 0, jump to loop
      move jabs_cond, #loop;
```

Even if not impossible, the above code takes time to write and is difficult to read. This can be explained by the fact that, while seeing the program as displacements provides some advantages from an hardware perspective, considering the program flow as only *Move* operations requires a non negligible amount of adaptation from the programmer.

To counter this situation, we added to the assembler the possibility to define *macros*, that is, rules which transform a single statement into a sequence of instructions so that programming becomes less weary. These macros can use an arbitrary number of parameters and calls can be present in their definition. Using macros creates distance from the architecture model and allows the creation of a more practical *meta-language* that is based on displacements only but whose syntax is closer to traditional assembly programming. Of course, this approach hides somewhat the fact that every operation consists of data displacements in the architecture, but the flexibility gained for the programmer is worth

it, especially since the meta-language does not hinder the usage of a more low-level perspective if it is required.

Without macros:

```
begin:
    move jump_condition, #1;
    move jabs_cond, #after ; // Shall normally jump

    move assert_line, #current_line;
    move assert_a_t, #1 || move assert_b, #2; // Hangs if here

after:
    move adder_b_t, #1;
    nop;
    move r0, adder_r;
    move cond_op, #zero || move cond_a_t, r0;
    nop;
    move jump_condition, cond_result;
    move jabs_cond, #dead_end;
    move assert_stop, #0;

dead_end:
    move assert_line, #current_line;
    move assert_a_t, #1 || move assert_b, #2; // Hangs if here
```

With macros:

```
#include "macros.inc"

begin:
  jump after; // Shall normally jump
  error; // Hangs if here

after:
  add #0, #1;
  jz add_result, dead_end;
  exit; // Everyhing's ok

dead_end:
  error;
```

Figure 4.3: Code sample written with and without macros.

Defining a meta-language proved very useful and effective, notably from a programmer's perspective. Figure 4.3 shows the same program implemented once with macros and once without.

It was thus possible to define function calls (with context saving on the stack) in a few lines of codes. In addition, the programmer can define additional complex macros that can be used for achieving even higher-level operations (such as loops, complex tests, ...).

4.1.5 Code and data segments

The assembler offers the possibility to define various memory segments that refer to different memory locations to store instructions and data, the first being defined by the `.code` keyword and the other by the `.data` keyword. Even if different segments can be declared in the code by using the keywords in multiple locations, the segments are gathered when generating the code so that the data segments immediately follow the code segments (which start at address location 0).

To declare a data segment (i.e. variables stored in the memory), the `.data` keyword must be used. It can be optionally followed by the initial values of this code segment, declared by a label. The following example shows the process of declaring an array of 10 values with an initial content along with a 512-word zone with an initial value of 1 that can be accessed with the address *vector_1*:

```
.data
 array:
   .dw 16,12,14,14,18
   .dw 11,12,13,17,22

 vector_1:
   .size 512 1

.code
 [...]
```

Listing 4.1: Declaration of a data segment.

4.1.6 Arithmetic expressions

In assembly, constants that depend on arithmetic expression are sometimes required. This is the case for example when copying a memory zone. Let us now consider the following example, that uses the embedded memory controller, which proposes different addressing modes (see section 3.7.1):

```
move mem_op, #autoinc; // Use the auto−incremented addressing mode
move dpa, #table_1; // Memory pointer = table_1
move r0, #64; // Copy 64 elements

// While R0 ≠ 0, copy from a FIFO to table_1
loop:
  move memory_a, fifo; // Copy from a FIFO to table_1
  subr r0, r0, #1; // Decrement loop counter
  jnz r0, loop; // Loop counter = 0 ?
```

Listing 4.2: Non optimized array copy.

When arithmetic expressions are available in the assembler, the above code can be rewritten by using the expected final value of DPA (computed with an arithmetic expression) as a test condition:

```
move mem_op, #autoinc; // Use the auto−incremented addressing mode
move dpa, #table_1; // Memory pointer = table_1
move r0, #(table_1 + 64); // Maximum value, using an arithmetic expression

// While DPA ≠ R0, copy from a FIFO to table_1
loop:
  move memory_a, fifo; // Copy from a FIFO to table_1
  jneq dpa, r0, loop;
```

Listing 4.3: Optimized copy of an array.

With this code, the subtraction in the loop can be removed, which in this case translates to a reduction of 3 move instructions from the loop (of the 8 initial instructions).

Arithmetic expressions can be used anywhere in the code and can be composed of an arbitrary number of parentheses. Labels can also be used in expressions, however only if they have already been declared when used[3]. This can be explained by the fact that expression resolution is done during the first pass of the assembly process and not during the parsing of the tree, for the sake of simplicity. However, by that time the position of every label is not yet determined. Even if this approach might be limited, it still proves useful.

The operators that can be used in the expression are the following[4]:

$$\mathrm{Op} = \{+, -, *, /, \ll, \gg, \vee, \wedge, \oplus\}$$

With these expressions, masks can be realized as well as label relative operations, loop size computations..., without having to use constant values in the code, which in any case is a bad programming habit. Consequently, programs are more readable and more parameterizable, two characteristics that help reducing errors.

4.1.7 Immediate values and *far jumps*

In this section we will tackle the question of *far jumps*, that is, jumps in a program to an address that is beyond the zone addressable by immediate values. It was stated in section 3.4.2 that immediate values use 16-bit values in the standalone and cellular versions whereas the VLIW version uses 9-bit immediate values. This leads us to the definition of two kinds of immediate values: the *short*, which can be represented completely in an instruction, and the *long*, which exceeds the 16- or 9-bit limit. This

[3] If it is not the case, an error message is generated by the assembler.
[4] The syntax is the one of the C language.

differentiation is required whenever as the program is big enough that jumps beyond the immediate value limit are required. As it is not reasonable to ask the user to avoid jumping too far or to see if the generated code is correct, the assembler had to take these situations into account.

The solution is theoretically simple:

1. During the parsing of the tree, when a node whose value can not be represented by an immediate is discovered, multiple instructions have to be generated to separate this value in smaller pieces that fit the instructions. The immediate value so cut is thus displaced to the concatenation unit[5] to form a bigger value. For example, completing the following operation:

   ```
   move r0, #0xabcdef12;
   ```

   requires to generate the following code to fit the instruction format:

   ```
   move concat_0, #0xef12; // Low 16 bits
   move concat_1, #0xabcd; // High 16 bits
   move r0, concat_r; // Move result
   ```

2. The same solution has to be applied to labels, as they are also stored as immediate values.

The use of long immediate values implies that not one but many lines of code have to be generated. This implies that a further assembly pass is required in order to resolve labels correctly.

To assign addresses to labels, the labels are annotated with the current number of generated lines when encountered in the code. However, when a label is *used* in the code, one must consider that if its address can not be reached by a short value, multiple instructions have to be generated. This has the consequence to invalidate all the labels situated after the line where the immediate had to be expanded, because the number of lines of the program has been changed. To reflect the new situation, all the affected labels have to be incremented by the number of lines generated by the expansion.

An example makes this concept clearer:

```
label_1:
   jump far_label; // Jump to a yet unknown location

label_2: // label_2 value := 2
   move r1, add_r;

[...] // more than 0xFFFF lines for cell and standalone revisions , 0x1FF for the VLIW version

far_label:
   move r0, #1;
   jump label_2; // label_2 value used here is wrong (≠ 2), as some lines will be added by the immediate
       extension
```

Listing 4.4: Far jump example.

During the first pass, the `jump far_label` instruction will generate a node with an unknown value, which is normal. However, during the rest of the pass, the label *label_2* will be declared as located at line 2, which is correct if *far_label* is not located outside the zone addressable by a short immediate, but at that moment we have no way to know it. As we continue parsing the tree, a jump that uses the concatenation unit will be done, which will imply that the *label_2* and *far_label* need to be changed, invalidating the program.

An elegant solution is provided by an additional assembly pass, for the following reasons:

1. The purpose of the *Visitor* design pattern is to realize different functionalities on a given tree.

[5] Note that we verify during assembly that such an unit is present in the processor by checking if the corresponding operands are declared.

2. By using a visitor to modify the tree, the other visitors (such as the code generator visitor) do not need to be modified, which simplifies development. Code modularity is thus preserved thanks to an appropriate pattern usage.

Thus, before generating code, a visitor parses the tree and updates the label values with their respective position whilst modifying the references to these labels if required. Figure 4.4 illustrates this new phase in the assembly process.

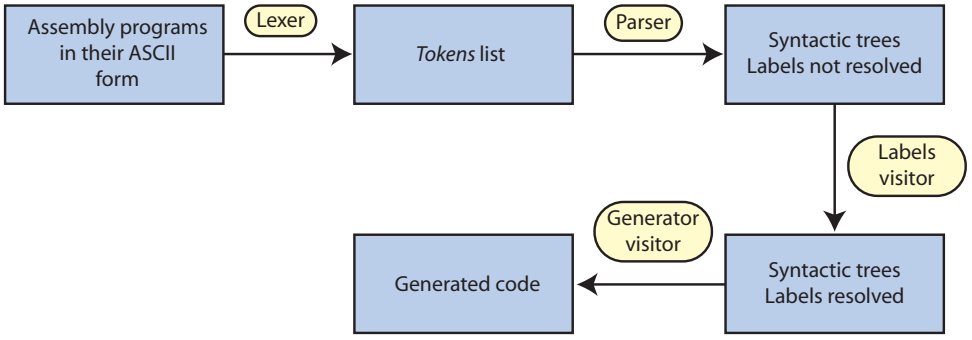


Figure 4.4: Complete assembly process with labels resolution.

4.1.8 Miscellaneous items

To conclude with the assembler section, here are some further remarks:

- It is possible to define the maximum stack size (`.stack_size`) and the total memory available to the processor (`.mem_size`) so that the assembler can check if the program fits the memory.
- Among the various parameters that the assembler can handle, the *verbosity* level can be changed to handle different statistics regarding the program. An example of such an output can be found in the annex, section A.4.

4.1.9 JavaCC

To implement the assembler, we used *JavaCC*[6]. This program generates the *parser* code directly from the language grammar. This solution was chosen because:

- Programming a *parser* is a fastidious task, and we preferred using a method that provides a better code reusability.
- Using a code generator enables to change the language grammar rapidly, which is not so easy to achieve with a classical program. In our situation, it was useful to quickly add extensions that were not initially planned.
- *JavaCC*, unlike better-known programs such as *Flex* or *Bison*[7], generates *Java* code and not C. Because the programs obtained are more portable than with C (Java "write once, run anywhere"), their use on different operating systems is easier.

[6]Information on that program can be found on `https://javacc.dev.java.net/` or [Succhi 99]. Note that this software, initially developed by a *Sun Microsystems* team, is now available under the *BSD Open-Source* (`http://www.opensource.org/licenses/bsd-license.html`) license.

[7]See `http://dinosaur.compilertools.net`

- An additional module called *JJTree* simplifies the generation of syntactic trees.

In conclusion, this is a very powerful tool with the drawback that the documentation and the examples are a bit light (notably on how to implement complex operations). Also note that other similar software exist such as *AntLR*[8] or *JFlex*[9].

4.1.10 Language grammar

The BNF form of the language is provided in section A.5 of the annexes, p. 227.

4.2 GCC back-end

We saw in the previous section that assembly programming on a TTA can sometimes be difficult or tricky, notably because the programmer must think in terms of *data displacements* whereas it is most common to think in terms of *operations*. If the introduction of macros in the assembler simplified programming, the development of complex programs using assembler remains tedious.

To overcome this difficulty, a compiler for high-level languages was required. Another argument in favor of the adoption of a compiler resided in the fact that most of the existing algorithms available in the public domain for such a processor (such benchmarks or multimedia code) are written in programming languages such as C. Their port would require a complete rewrite to be used on the ULYSSE processor, a situation that would render the CPU if not useless at least crippled. The final point in favor of the development of a full-fledged compiler was to improve its chances to be accepted in a near future as a solid tool for research projects (the processor IP sources along with the compiler and all the necessary tools for using it will be released, using a GPL license, in the OPENCORES [10] community).

The choice of the language to be supported seems relatively evident in the context of embedded systems. In fact, the C language is the most serious candidate considering the traditional use of this language and with respect to what has been done in the field so far. Keeping that in mind, we then had to decide how the implementation of the compiler itself would be done. The language being chosen, we had two main choices to implement the compiler, the first being a development starting from scratch, the second consisting in porting the back-end of an existing compiler for the ULYSSE processor.

4.2.1 Why GCC ?

Preliminary remark on GCC *The following explanation of why GCC was chosen is partly based on the work of Michel Ganguin [Ganguin 08], whose master thesis was on the implementation of the compiler. We will also use some of the elements described in the same document when we describe GCC internals (section 4.2.2) and function calls (section 4.3).*

We decided early not to begin the development of a new compiler, the flexibility offered by a complete development being largely outweighed by the complexity of the task. Luckily, several compilers offer programming helps to add new targets using separated source codes for the *front-end* and the *back-end* parts, the former being responsible for the transformation of pure C code to an adapted data structure that can then be processed, the latter being used to generate the corresponding assembly code. Among the different choices, the *GNU C Compiler* (GCC) [Gough 04] seemed to be the most promising candidate because its popularity ensures a large developer community support and because the existence of ports for many different architectures provided plenty of examples to start with. Two other candidates were considered: LCC [Fraser 91] and SDCC [Dutta 00]. The first would also

[8] `http://www.antlr.org/`
[9] `http://www.jflex.de/`
[10] `http://www.opencores.org`

have been appropriate but it was finally rejected due to a more limited documentation. The second, SDCC is aimed more at very small processors that work with less than 32 bits.

4.2.2 The GNU C Compiler – GCC

We followed the incremental approach of writing a back-end described by [Deshpande 07] but also in the *"Workshop on GCC internals"*[11]. We will not go into too much detail here because writing a compiler mainly consists of implementation-related issues. However, a complete "walkthrough" describing the whole process can be found in Michel Ganguin's master thesis report [Ganguin 08].

4.2.3 GCC components

GCC is more than "just" a compiler as it gathers together a preprocessor, an assembler program, a standard library for linking and, obviously the compiler itself. When GCC is called in a standard way, as in:

```
gcc [args] cfile.c -o program
```

In reality, four programs are used to generate the final binary executable code:

```
cpp -I/stdheaders [cpp specific args] cfile.c -o tmp.i
cc1 [cc1 specific args] tmp.i -o tmp.s
as [as specific args] tmp.s -o tmp.o
ld [ld specific args] -L/stdlibs -lc -lgcc tmp.o -o program
```

- `cpp`, the C preprocessor, handles preprocessor directives, such as includes and macro expansions. Some of the GCC arguments are passed to `cpp`, such as the `-D` flag, to define a macro[12]. Some target-specific macros are included, i.e. in this port the `MOVE` macro is defined (usable for example to produce ULYSSE-specific code with `#ifdef MOVE`). `cpp`'s output is a C program with macros resolved, comments removed, etc. It is part of GCC.
- `cc1`, the compiler, contains the whole compiler, front-end, "middle-end" and back-end. It produces textual assembler code. Obviously, `cc1` is part of GCC.
- `as`, the assembler program, produces binary code. It is not part of GCC but is a script that call the real assembler program of the processor or may be a symbolic link to it.
- `ld`, the linker, brings together several assembled programs and allows the inclusion of libraries.

As we already have created the assembler program, we focused our attention on the `cc1` part. We did not develop a linker, which means that the input, after preprocessing, has to be one single file (which can be achieved by adequate programming techniques).

4.2.4 The compilation process

When we consider the whole compilation process (Figure 4.5), GCC makes use of three intermediate languages:

- Starting from the source language, a front-end transforms the source code (e.g. C) into GENERIC, a high-level tree-based language in which some source-language specifics may remain.
- The second language is GIMPLE, a low-level tree-based language, more restrictive than GENERIC, but still source-language and target machine independent. GCC can perform several optimizations directly on the GIMPLE form of the program (see 4.3.2).

[11] More details, presentation slides and code samples are available on `http://www.cse.iitb.ac.in/~uday/gcc-workshop`

[12] Note that C macros are different from assembly macros and are handled by the preprocessor in this case.

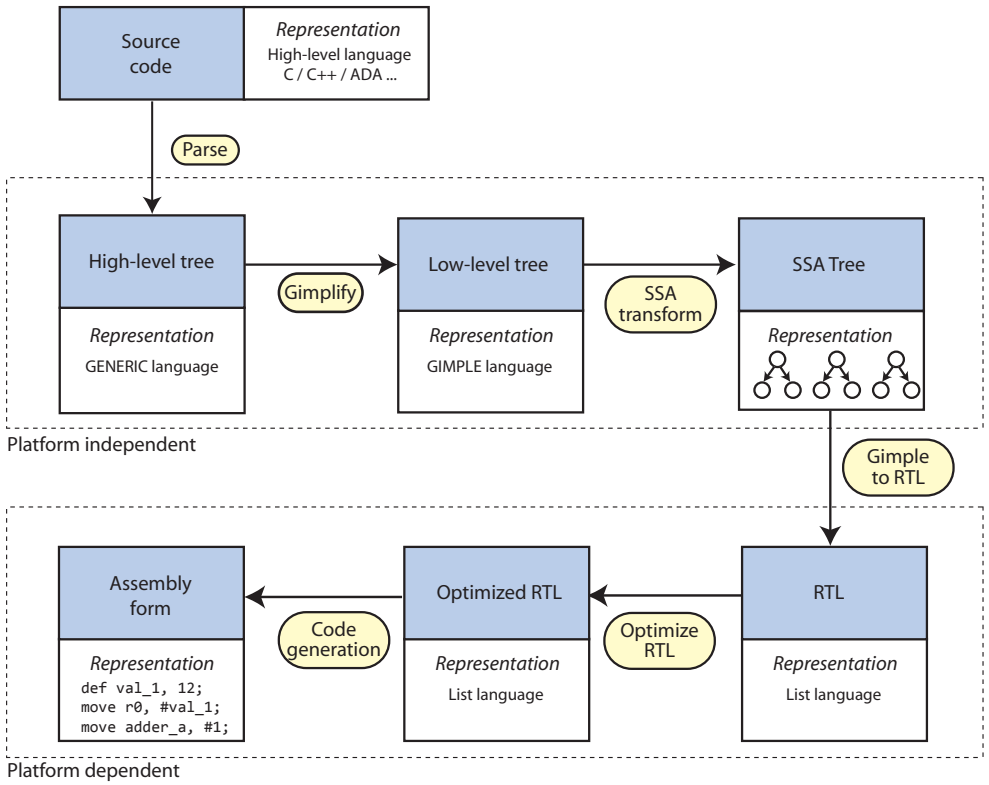


Figure 4.5: The compilation phases.

- The third language is RTL, which consists in a stream of instructions that are transformed, in the last pass of GCC, directly into assembler.

Writing a back-end is a twofold process. Firstly, the underlying mechanisms of the machine description file, i.e. how instructions and constraints (number of registers, read–write access to register, memory layout, ...) are defined, must be understood. Secondly, the code generation process has to be performed correctly.

In our opinion, the way the instructions are described within GCC is quite effective as it collects together different components, used at different locations in GCC's source code, to minimize code redundancy and to maintain a good overview of the processor. The GIMPLE to RTL translation is also very well done: once the register constraints are correctly written, translation is transparent. As a result, this first phase of the compilation is relatively straightforward.

The next phase, which works on the RTL representation, is more complex as new instructions are introduced to match the constraints of the architecture. This representation starts with an unlimited number of registers. Several optimization passes that could not be done earlier can then be performed (see 4.3.2). Finally, register allocation is done, spilling to memory when required.

After this phase, the assembler code can be generated, based on the optimized RTL representation.

4.3 Handling function calls

In this section, we will explain how the functions are called with GCC. As function calls do not depend only from the hardware, several implementation alternatives can be chosen. We have decided to structure memory as depicted in Figure 4.6, with the code and data segments located at address 0 and the stack growing downwards from the end of the memory.

ULYSSE does not provide a specific method to call functions, in the sense that calls are handled as "normal" jumps. For this reason, a *prologue* and an *epilogue* must be added to jumps to make them

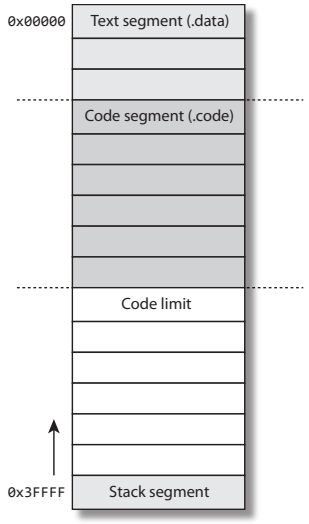


Figure 4.6: Memory organization as seen by the compiler.

behave like functions calls. The major difference between *jumps* and *calls* is that the latter suppose that the control flow is resumed at the end of the call.

4.3.1 Saving state

To achieve this, we have at our disposal a specific register (`RET`) in which the caller can save the return address before performing the call. From the callee's perspective, there is only need to jump to this address at function's return.

Another piece of data that need to be saved are the arguments passed to a function and the return value (in GCC, the latter can be considered as an additional argument). Therefore, for a non-void function, the return value becomes the first argument and every nth argument is considered $(n+1)$th, as shown here:

```
// Function definitions can be considered as
int foo(int a); -> void foo(int *retval, int a);

// Function calls are transformed to (not ANSI C, but correct in GNU C)
int res = foo(43); -> int res = {int retval;
                                 foo(&retval, 43);
                                 retval;}
```

All the function arguments are passed on the stack, in the order they are declared.

In short, most of the work is done by the caller (saving the return address and the arguments, changing the stack pointer) and it cannot leave values in registers across a call, as no register can be considered safe[13]. For the callee, all arguments are put on the stack and the return address is placed in the appropriate register.

4.3.2 GCC optimizations

One of the advantages of using a well-supported compiler such as GCC is that many optimizations are directly available. Notably, GCC provides optimizations at different language levels, as we will now examine.

[13]Which means that to hold a value over a call, it must be stored into memory before the call.

GIMPLE-level optimizations

In this language several optimizations are performed (see [Stallman 07, section 8.4]). These optimizations are often called the *"middle-end"* of GCC, for they are source-language and target-architecture independent. These optimizations include techniques such as:

- *inlining*, which consists in replacing the call to a function with its body;
- *constant propagation*, which consists in replacing constants or variables that are never modified by immediate values so no memory operation has to be performed;
- *transformation into static single assignment* (SSA) form, to simplify many aspects of code analysis.

RTL-level optimizations and operations

RTL is a language inspired by the *Lisp* list and the first form of RTL produced from GIMPLE uses an unlimited number of registers. After a second transformation, each RTL instruction corresponds to an assembler instruction (it is said to be in "strict RTL" form). Several optimization passes (mandatory or optional, and which could not be done earlier in GIMPLE) are done on this language (see [Stallman 07, section 8.5]). Some of these are listed here:

- *Control flow graph cleanup*
  Examples are dead code elimination that removes unreachable code, jump simplification (to transform double jumps into single ones), etc.
- *Common subexpression elimination*
  To compute only once a common subexpression found in several different expressions.
- *Liveness analysis*
  Locates the first and last use of variables so determine the liveness of a variable. If the liveness of two variables overlaps, two different registers are needed to store them, but only one when they do not overlap. This analysis is useful, as in the former example, for register allocation but also to locate unused variables.
- *Instruction scheduling*
  When there are different latencies for operations, instead of waiting for the end of a "long" operation, it is sometimes possible to execute further instructions. This implies that the program is executed out-of-order.
- *Register allocation*
  Finds the best available register candidate and, if there is none, uses a stack slot on memory. The frame pointer can be eliminated in this pass, by computing its offset from the stack pointer. This elimination is mandatory for the ULYSSE processor, because of the limited number of registers that can be used to directly access memory with an offset (see section 3.7, page 38).
- *Final pass*
  Generates the assembler code by translating each strict RTL instruction into its equivalent assembler form.

4.3.3 ULYSSE peculiarities for GCC

For some aspects of the compilation, the non-conventional nature of ULYSSE presented difficulties during the port of GCC for the architecture. Even if the purpose of this chapter is not to provide a detailed description of the structure of the GCC back-end and of the required implementation changes that were undertaken to tailor it for ULYSSE, some aspects of its development are worth mentioning to illustrate the difficulties introduced by the use of non-conventional processors.

In fact, because GCC was initially designed for standard RISC/CISC processors in which operations are not simple displacements, it was necessary to tell the compiler that moves into the trigger

registers had to be seen as the operations themselves. Because of the special semantics of the move operation, we had to inform GCC that some registers trigger operations, i.e. that a displacement to these registers should not be considered as moves but as the operations themselves. If a read operation of a register triggers an operation, it is also necessary to tell the compiler that the corresponding operation has a double effect (this is not implemented in our code because all our trigger registers are read-only, see section 3.2.2 page 29.

Moreover, further problems were caused by the fact that some registers are *read-only*, such as the registers holding results of operations. For GCC, every register is expected to be fully accessible when performing some of the optimization passes. Furthermore, due to the number of constraints on each register (readable or not, etc.) in the processor, GCC has sometimes trouble finding solutions for register allocation.

Finally, we also had to face the fact that the GCC heritage can sometimes be problematic. For instance, the fact that ULYSSE uses a full 32-bit approach even for memory access (data words are not aligned on *bytes* but on 32-bit words) was a source of multiple bugs during the development process, GCC being relatively lax on its definition of bytes, words and other sizes.... As a result, some constants that were supposed to automatically reflect different sizes had to be changed by hand because they were implicitly supposing that accesses were done on 8-bit words.

4.3.4 Compiler optimizations specific to TTA

Although the implementation of a compiler for a TTA processor presented in this work remains simple in the sense that a *reliability* objective was preferred over strict performance, optimizations specific to TTA can be undertaken. Even if these optimizations were not applied in the present GCC port, they are worth mentioning because they provide interesting opportunities for improvement.

The most important contribution in the domain is the seminal work that was done by Hoogerbrugge for his PhD thesis [Hoogerbrugge 96]. Notably, he proposed several kinds of optimizations. Let us now consider the following code[14]:

```
add r1, r2, r3; // r1 = r2 + r3;
sub r4, r2, r5; // r4 = r2 − r5;
store r4, r1;  // mem[r4] = r1;
```

Listing 4.5: Code sample to optimize.

The direct translation of each of these n-operand, m-result operation results in the following $n+m$ `move` operations:

```
move add_a, r2; // first addition operand
move add_b_t, r3; // second addition operand
move r1, add_r; // move addition result to r1

move sub_a, r2; // first subtraction operand
move sub_b, r5; // second subtraction operand
move r4, sub_r; // move subtraction result to r4

move store_a, r4; // store address = r4
move store_v_t, r1; // mem[store_address] = r1
```

Listing 4.6: Translation to `move`-only operations.

This straightforward translation offers several optimization possibilities, notably if multiple displacements are allowed. Before examining them, let us now define the concept of schedule in the context of TTA:

[14]This example is based on the explanation made in [Corporaal 99, p. 19–20] on the same subject and follows the same methodology, although with a different notation.

Definition A *schedule* of a program is the temporal assignation of the corresponding data displacements among the different FUs of the processor using the existing move slots.

Using this definition, let us consider an example where four move slots are present in a processor along with two separate arithmetic units, namely one for the add and one for the sub.
A possible *schedule* for the program 4.6 is the following :

Cycle	Move slot 1	Move slot 2	Move slot 3	Move slot 4
1	r2 → add_a	r3 → add_b_t	r2 → sub_a	r5 → sub_b
2	add_r → r1	sub_r → r4		
3	r4 → store_a	r1 → store_v_t		

If we consider now the possibilities to optimize this code, several opportunities arise, namely:

1. **Data bypassing**
   Because the arithmetic results are present at the output of the two arithmetic units, they can be moved directly to the memory unit without prior displacement to the `r1` and the `r4` registers. This allows to spare one cycle. The new scheduling is then:

Cycle	Move slot 1	Move slot 2	Move slot 3	Move slot 4
1	r2 → add_a	r3 → add_b_t	r2 → sub_a	r5 → sub_b
2	sub_r → store_a	add_r → store_v_t	add_r → r1	sub_r → r4

2. **Dead move elimination**
   If the purpose of the `r1` and `r4` registers is only to store the intermediary results for the store operation, i.e. they are not used anywhere else before being rewritten, they can be considered as *dead code* in the schedule and simply removed. In this example, this optimization does not improve the timing of the schedule but its memory requirements. Thus, the resulting schedule becomes:

Cycle	Move slot 1	Move slot 2	Move slot 3	Move slot 4
1	r2 → add_a	r3 → add_b_t	r2 → sub_a	r5 → sub_b
2	sub_r → store_a	add_r → store_v_t		

3. **Operand reuse**
   In some cases, operands can be reused if already present in a given FU, removing redundancy. For example, the following code:

```
add r1, r1, #1;
add r2, r1, #1;
```

   translates directly into the following `move`-only code:

```
move add_a, #1;
move add_b_t, r1;
move r1, add_r;
move add_a, #1; // Useless
move add_b_t, r1;
move r2, add_r;
```

   As the immediate value 1 is already present in the first operand of the adder, the marked instruction is useless. This is of particular interest in single move bus implementations such as ULYSSE where one displacement corresponds directly to one clock cycle and can be used notably to speed-up loops when adding offsets or increment values.

Even if it can not be considered as an optimization technique itself, TTA processors have the advantage of increased freedom thanks to the fact that displacements and not operations have to be scheduled. Thus, because of the trigger mechanism, operands do not have to be moved at the same time. Moreover, as many values can be stored directly into FUs without passing trough a centralized register file, the scheduling constraints are lightened. Both phenomena allow more room for scheduling improvements.

4.4 Conclusion

During the development of the software tools to support our processor, we considered *reliability* of the software suite as the most important factor of development. Of course, efficiency is also important but we are intimately convinced that a *working* design is better than a *fast* but unreliable experience, especially in our context where even the hardware was developed from scratch. Thus, because software is easier to test than hardware, we put a lot of effort in obtaining robust tools that produced *correct code*.

The goal of a reliable and universal compiler was attained. We will see in the remainder of this work that it will be used to compile thousands of lines of C code with negligible problems. Porting a compiler such as GCC is an interesting challenge that requires a considerable amount of work and it should come as no surprise that some problems and limits still persist:

- First, some optimizations are not always possible, a situation that limits the efficiency of the code. In most "real"[15] cases, optimizations have to be completely turned off, otherwise the compiler crashes. Many problems have been identified as responsible for this. Unfortunately, solving them was impossible due to time reasons. However, the code produced when no optimizations are used (which is done by compiling the programs with the `-O0` flag) is correct. Thus, the compiler can fail but no incorrect code is ever produced.
- Secondly, neither software floating-point operations nor the C standard library have been entirely ported yet.

For further work on the tools themselves, the instruction scheduling should be improved and, since the TTA architecture allows to easily add new operations, the back-end should allow this flexibility too. Moreover, even though the hardware may resolve hazard dependencies automatically thanks to the busy bit mechanism to ensure correct execution, the compiler should, when possible, take latencies of instructions into account to achieve best performance. Otherwise, FUs could unexpectedly stall and reduce performance of the compiled code. On a more anecdotal level, a linker and a standard library should be also developed.

Except for these limitations, the assembler and the compiler are fully working and the results obtained so far are very encouraging: they open the way to the realization of high-complexity applications on our bio-inspired substrate, as we will see in the next chapters.

[15] When the source code complexity is more than 10 lines

Bibliography

[Corporaal 99] Henk Corporaal. *TTAs: missing the ILP complexity wall*. Journal of Systems Architecture, vol. 45, pages 949–973, 1999.

[Deshpande 07] Sameera Deshpande & Uday P. Khedker. *Incremental Machine Description for GCC*. Indian Institute of Technology, Bombay, 2007. `http://www.cse.iitb.ac.in/~uday/soft-copies/incrementalMD.pdf`.

[Dutta 00] Sandeep Dutta. *Anatomy of a Compiler – A Retargetable ANSI-C Compiler*. Circuit Cellar Ink, no. 121, pages 30–35, Aug. 2000.

[Fraser 91] C. W. Fraser & D. R. Hanson. *A Retargetable Compiler for ANSI C*. Technical report CS–TR–303–91, Princeton University, Princeton, N.J., 1991.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1995.

[Ganguin 08] Michel Ganguin. *A GCC Backend for the MOVE Architecture*. Master's thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2008.

[Gough 04] Brian J. Gough & Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[Hoffmann 04] Ralph Hoffmann. *Processeur à haut degré de parallélisme basé sur des composantes sérielles*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[Hoogerbrugge 96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[Palsberg 98] Jens Palsberg & C. Barry Jay. *The Essence of the Visitor Pattern*. In Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC, pages 9–15, 1998.

[Stallman 07] Richard M. Stallman & The GCC Developer Community. *GNU Compiler Collection Internals. For GCC version 4.3.0*. Free Software Foundation, 2007. `http://gcc.gnu.org/onlinedocs/gcc.pdf`.

[Succhi 99] Giancarlo Succhi & Raymond W. Wong. *The Application of JavaCC to Develop a C/C++ Preprocessor*. ACM SIGAPP Applied Computing Review, vol. 7, no. 3, 1999.

Chapter 5

Testing and performance evaluation

"Beware of bugs in the above code;
I have only proved it correct, not tried it."

DONALD KNUTH, 1977

THE DEVELOPMENT of a complete processor, especially with a design starting from scratch, requires careful planning of the testing methodology to be used throughout the process. The ULYSSE processor is no exception and we will discuss in this section the methodology we followed during the various phases of design. We will also demonstrate that TTA processors also present some advantages when it comes to testing, thanks to the common unified interface to which each functional unit must adhere to.

Once the correct functioning of the processor is guaranteed to a certain extent, we will present and discuss some experimental results in section 5.5 to evaluate the processor's performance with a series of benchmark programs. In this second part of the chapter, we will also evaluate how the code generated by our compiler compares, in terms of size and speed, to another 32-bit processor. This comparison will serve as a confirmation that the implementation choices taken were reasonable, but also that our bio-inspired approach does not hinder performance too much.

5.1 Testing methodology

To define how the processor was tested, some definitions are required.

5.1.1 Definitions

The testing methodology is closely based on the granularity of each component of the processor. Because we used a bottom-up and very compartmentalized approach to build the processor, testing can follow the same methodology. To determine the extent to which a component can be tested, we define the following concepts:

1. **Definition** Two test vectors can be considered *equivalent* if and only if their application at the inputs of a component under test imply the same changes, in terms of internal state transitions, of the component. In other words, the *internal state machine* of the component follows the same transitions, if any, for both test vectors.

   To illustrate this concept of *equivalence*, let us now imagine the following situation, where the design of a 8-bit register must be tested.

Equivalence example

This component contains an 8-bit bus input for the data and three control inputs that represent the clock, reset and enable signals. The output of the device is also an 8-bit bus. To fully test this component, a normal test bench would need to check if all of the 2^{11} different inputs combinations produce the correct output. However, if we can ensure that, due to the nature of system under test, the eight data bits are not used in the determination of the next state, their role is different from that of the other inputs. As such, it is safe to consider *equivalent* the input vectors that only differ by their data bits. It is then possible to reduce the total number of test vectors to 2^3. Of course, this reduction is only possible with a careful examination of the design that limits its application to small designs, to avoid missing special cases.

2. **Definition** A *testability metric* ω is an estimate of the coverage of the total possible test cases by a given test bench. A testability $\omega = 1$ signifies that *every* state and input combination of an hardware component can be checked and asserted by a test bench. Similarly, a testability $\omega = 0$ indicates that nothing can be tested about a component[1].

   The testability metric can be used, in conjunction with other criteria, to define *testability classes*, a concept that helps classifying hardware components in different categories that can then be used to pinpoint the origin of errors more easily. In our context, we consider the following classes:

   (a) *Class C1* - Fully testable components
   Components for which every combination of inputs and states can be tested to guarantee functionality. To reduce the number of tests required, *equivalent test vectors* are used. The testability of these components is $\omega = 1$.

   (b) *Class C2* - Functionally testable components
   Components that can be tested to a point where it can be stated with a high level of confidence that the component's functionality is verified thanks to a sufficient number of tests, even if an exhaustive test bench is not feasible. Particular attention should be paid to cover corner cases.

   (c) *Class C3* - Complex components
   Components that require complex test benches, most of the time generated by programs because they imply a complex flow of operations. These components must be heavily asserted (see next section) to cover all the branches of the flow.

   (d) *Class C4* - Irregular components
   This category covers the components that are difficult to test because they possess a high number of inputs and internal states that can not be asserted easily. In addition to this, we put in this category every component driven by multiple clocks and/or that must synchronize across multiple clock domains.

   (e) *Class C5* - Black-box components
   These components have the particularity that they only expose their input and outputs signals but their inside circuitry can not be accessed. The ω value for this kind of components is difficult, sometimes even impossible, to estimate because the input vectors that can be applied to the inputs do not necessarily imply that every case is tested because of the internal memory states.

   Of course, ω is only an estimate and remains dependent on the designer's experience. In and for itself, the numerical value of the testability metric, besides the extremes, is not essential as long as it allows to determine the degree of confidence the designer can have in the *reliability* of the component. Thus, during the design and test of the processor, the existence of testability

[1] Note that a testability of 0 is a degenerated case meaning that nothing can be tested about a component. This could happen for example in the case of a component without outputs.

classes was primarily used to determine more quickly the order in which the components had to be checked to find an error.

We will now introduce two methods we used to control the validity of the different classes.

5.2 Testing with asserted simulation

Whenever possible, we introduced *assertions* in the code to detect design failures. According to [Foster 03, page 3], an assertion is

> "[...] a statement about a design's intended behavior (that is, a property), which must be verified. Unlike design code, an assertion statement does not contribute in any form to the element being designed. Its sole purpose is to ensure consistency between the designer's intention, and what is created."

Assertions should not be considered as components used to externally test components but rather part of the testing and development itself. During the development of the processor, three levels of assertion were used, based on the language the assertion was done:

- *VHDL level*
  At the hardware description level, two different assertion languages were used:
  - VHDL `assert` statements. This basic form of testing belongs directly to the VHDL language itself. It allows to check the consistency of simple assertions, such as in the following example:

    ```
    assert (data_width - 1 = 31) report "IP␣Multiplier␣was␣generated␣with␣32␣
        bits␣operands.␣You␣are␣using␣a␣different␣data␣width." severity error;
    ```

  - Property Specification Language (PSL) statements. This language was recently developed [Foster 05, Eisner 06] to provide an assertion method both for verification and simulation [Boule 05]. It provides a lot more flexibility than standard VHDL asserts because time and state relationships can be checked. A simple example such as

    ```
    psl assert never (nCS0 = '0' and nCS1 = '0') report "Two␣nCS␣selected␣at␣
        the␣same␣time";
    ```

    would be more complex to realize with standard assertions, requiring intermediary signals to achieve the same behavior. More complex example can also be considered:

    ```
    -- Make sure that the address won't change when fetching new data
    psl assert always ((rst = '0' and fetch_mem_req = '1') -> next(prev(
        fetch_mem_address) = fetch_mem_address))@(rising_edge(clk))
       report "Address␣not␣stable";
    ```

  Both languages are supported by the simulator we are using (Mentor *ModelSim SE PLUS version 6.3f*[2]) and assertion failures are clearly indicated in the simulator. Depending on the severity of the assertion failure, the simulator can be stopped or simply textually indicate that an assertion failed.

- *Assembler level*
  The goal of asserting assembly code is to guarantee that certain statements always hold. Situated at a higher level than VHDL, assembler provides the first abstractions that enable behavioral checking. In addition, assertions are also used in assembler as a method to report information to the simulator or even to modify the simulation itself. These two cases are illustrated in the following code:

[2] `http://www.mentor.com`

```
// ALU test bench
#include "defs.inc"
#include "macros.inc"

.code
addr r0, #12, #2;
assert_eq r0, #14;
subr r1, #12, #5;
assert_eq r1, #7;
[...]
move r7, #0xbadface;
assert_eq r7, #0xbadface; // Test long immediate
exit; // Stop simulator
```

To be able to interact with the simulator, we simply added a non-synthesizable FU for assertions (see appendix page 209). Displacements to adequate registers can be used to trigger a VHDL assertion to stop the simulator and to report errors. In addition, it is also possible to report the line of the current instruction of the code in assembler (`#current_line`). To illustrate how this assertion mechanism can be used to determine more easily source code errors, we ran the above code with a faulty subtraction unit that always return the value 10. The output of the simulator is the following:

```
# ****************************************************************
# *** ERROR Line 12
# *** Assertion unit equality fails : 00000007 != 0000000A
# ****************************************************************
```

With no error inserted, the `exit` statement at the end of the above code would have stopped the simulator differently, as shown in the appendix page 228.

- *High level language (C)*
  This level holds the most flexibility to insert assertions because it builds on the other levels, which implies that it can use all the assertion facilities of the assembler. In addition, the C language also defines its own assertion mechanisms that can be used in conjunction with the underlying mechanisms. The following code snippet (used to measure the execution time of a program) demonstrates how it can be used to check the validity of the results:

```
void main(){
int i, beginTime = 0, endTime = 0;

get_timer1_us(beginTime);
  quantize(block,8,0,1,32768);
get_timer1_us(endTime);

// Triggers simulator error if untrue
sim_assert_eq(block[0] == 87, 1);

// Exit simulator gracefully
sim_exit();

// If we get to this point in simulation, something wrong happened
sim_error();
}
```

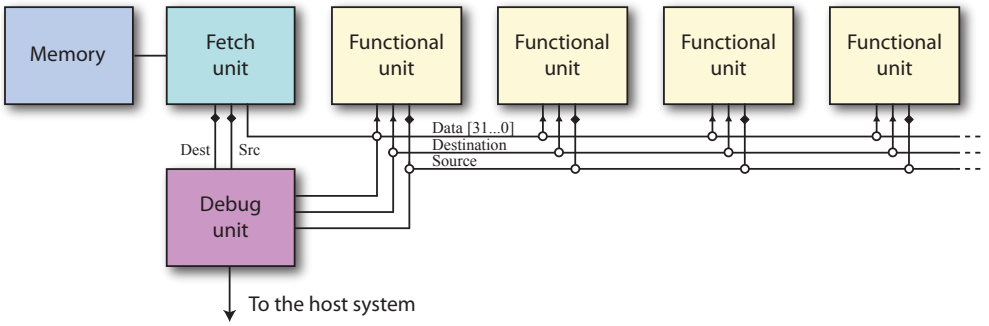


Figure 5.1: Processor schematic with debug interface.

5.3 In-circuit debugging

The other testing technique we used in our design leverages the fact that every functional unit reacts to the destination and source buses on which the decoded instructions are placed. By temporarily replacing the fetch unit on these buses (Figure 5.1), it becomes possible to use all the FUs as independent modules. The debug unit represented in the same figure contains a few registers, that can be read or written from an external interface, offering a direct access to the FUs. This allows notably:

1. To read the content of every readable FU register (input and output). The whole *internal state* of the processor can thus be accessed and sent to the user to visualize the state of the system.

2. To modify the internal state of the processor by writing in the registers, for debugging purposes. It is thus possible to see the effects of the modification of a register on a program, which can be useful during the development of programs.

3. To use the FUs without going through the fetch unit. This offers the possibility to realize computations but also, in the situation where the processor contains an I/O interface, to interact with external hardware. This can be interesting for example in the case where the processor interfaces a display, which can then be accessed from the CPU and from the debugging interface in a cooperative manner. Another possibility is to use the FUs only as hardware accelerators that can be easily accessed from a host computer.

The first point deserves a particular attention: whereas a standard processor requires to specifically plan mechanisms (such as a JTAG interface[3]) to enable access to internal registers, in the case of a TTA processor the situation is much simpler. By taking into account this specificity, we have developed a *debugger*, similar to what many development environments offer.

This tool can be used during the development of programs by providing a faster and simple way to examine execution, compared to simulation tools. However, it does not allow to examine more fine details at the cycle level such as timings or fetch mechanisms.

The debugger itself can be called *in-circuit*, which signifies that it does not simulate but makes use of the finished processor inside a FPGA. It allows to check that the outcome of a program is what is expected but also to verify that the FU *in silico* behave like in simulation.

At the hardware level, starting a debugging session corresponds to asserting the *halt* signal so that the processor goes to a state where the fetch unit releases control of the interconnection network and behaves as a "slave" unit like the others. It is then possible to control the internal buses as would the fetch unit and thus read or write every register in the functional units by putting the corresponding addresses on the buses. This mechanism can be used not only for debugging but also to modify the contents of the memory.

[3]See `http://www.jtag.com/`.

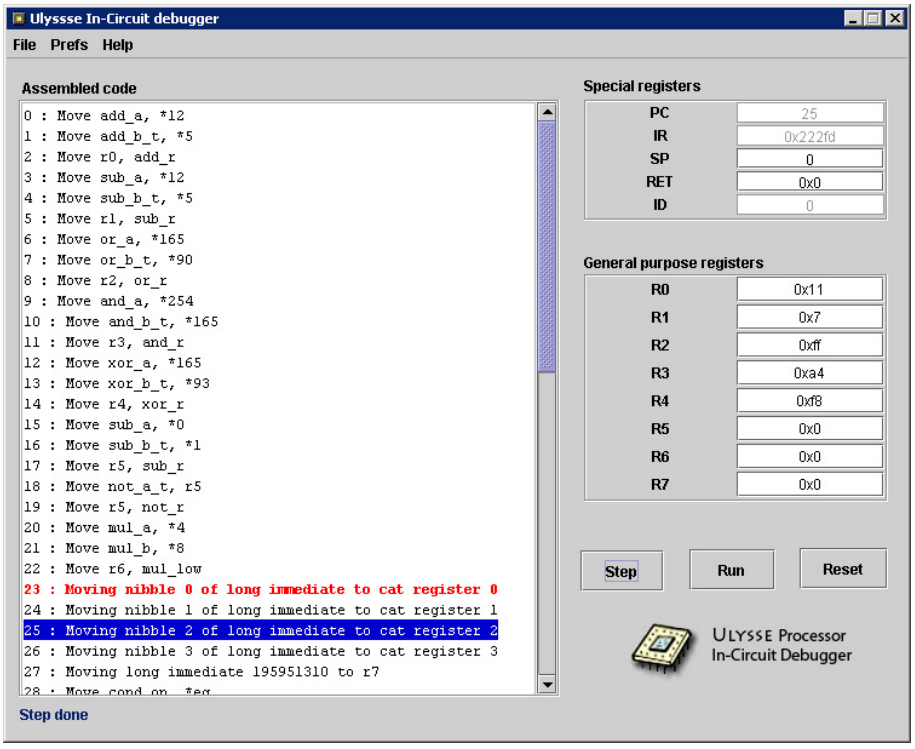


Figure 5.2: The debugger in function.

At the software level, the debugger consists of a *Java* program that communicates with the FPGA via a communication library. Its user interface, shown in Figure 5.2, resembles a standard debugger. Only a few registers are visible on the interface (so it is not too loaded) but every register can be changed. Their modification can be done by entering a numerical value (in decimal or hexadecimal), which in turn will modify the physical register[4].

The loaded program can be executed step by step or continuously, the user being able to insert breakpoints where the execution stops so to verify intermediary results. Many extensions can be imagined: taking into account the assertions present in the code[5], displaying and modifying the memory contents, ... Even if many improvements could be devised, this program remains interesting as is, not only as a proof of concept but also to verify the processor's behavior *in silico*. Finally, it also shows that the architecture presents some advantages for the development of a debugging interface thanks to a simplified access of its internal state.

5.4 Hierarchical testing of processor components

When applied to the ULYSSE processor, debugging and assertions can be applied to the testability classes taxonomy at different levels (from the bottom to the top):

1. **Component level – Testability class C1**
   Where the basic shared components used in the processor, such as counters or registers, are tested. The testability of these component can be considered as high and standard test benches

[4]We refer the interested reader to the debugger source code which has been realized by modeling the whole system as objects (registers, processor, ...) so it is also possible to extend the classes to realize a software simulator, which was not done in this work.

[5]They are used only during simulation right now. However, it is not required to remove them for a real target because the assertion unit is not synthesized, which translates an assertion in a series of `nop` operations on the processor.

that use equivalent test vectors and assertions can be easily used. In our implementation, the following components have been tested using a dedicated test bench:

- generic width counter with settable count value;
- generic width shift register with write enable signal;
- generic width register with settable initial value;
- the bus interface.

2. **The functional unit level – Testability class C2**
   Where the behavior of the different functional units is tested separately. Standard test benches can be easily used, with the advantage that every functional unit can be plugged into similar test benches as they all have the same interface. If possible, for FUs that include other VHDL components (such as IP modules), the external module was tested separately before being integrated. This was the case for example for the SRAM controller (section 3.7), which was tested with asserted simulation and implemented independently to conduct functional testing. The equivalence of test vectors can also be used to reduce the complexity of the test vectors.

3. **The processor level – Testability class C3 / C4**
   Where the global hardware behavior of the processor is tested. Testability at this level becomes more complex because it requires code to execute, which also means external memory loading, bringing the code generated by the assembler into the simulator, simulation of the external memory with timings and CPU interactions ... Equivalent test vectors are of little use to reduce the complexity of the tests but other debugging techniques can be used.

   Thus, debugging is done using the MODELSIM™ simulator where all the internal and external signals of the processor can be examined (see annex, page 229, Figure A.1). In addition to manual examination of the results within the simulator, this level is also checked with asserted simulation at all language levels along with debugging.

   This translates to a whole test suite written in assembler and C that contains test code for different functionalities. This test suite is first checked in simulation, using a command-line environment based on the MODELSIM™ interface. Because the GUI of MODELSIM™ is absent, simulations runs faster but at the expense of limited debugging possibilities, the only messages printed being those of the assertions in the code. A typical run of such a simulation is shown in annex page 228.

   The same testing suite can also be generated for an implemented version of the processor to check validity of the translation from the simulated model. With this suite, regressions are thus simplified and the correctness of new features can be asserted, following the idea proposed in [Meszaros 03] where systematic tests are used to validate code.

   As a side remark, we can also mention that test benches on the real platform were developed so that physical interactions with the hardware are not required and code downloads fast. Thus, testing code only requires a key combination in our development environment, saving time during debugging.

4. **The routing level – Testability class C4**
   Networks On Chip will be introduced in a further chapter (section 8.3) and we only mention the problematic here. Later in this work, tens of ULYSSE processors communicate together using a switched network that dynamically creates communication paths. Testing such a setup requires careful planning and is difficult, notably because of the highly irregular patterns that may appear, which imply that very complex test vectors have to be generated. Moreover, because the network spans different clock domains, clock relationships must also be considered. Achieving a good testability at this level is complex and requires considerable computational efforts (see for example [Pande 05] or [Coppola 04] and section 8.3.2).

One advantage of using testability levels resides in the fact that it permits to handle the increasing complexity of each level because it builds atop a functionally working module. Of course, the implementation of tests at each level requires a consequent time investment that, in our opinion, is largely

compensated by the fact that it becomes then possible to track problems very efficiently because the different layers where the error can reside can be handled separately.

5.5 Processor validation, performance and results

Once a correct processor behavior was achieved, the next logical step was to evaluate and test the processor's performance. To do so, several metrics that rely on different perspectives are possible:

Performance metrics To this category belong figures such as CPI, MIPS, Multiply-ACCumulate per second....

Single program tests This category contains figures that relate to standard program execution metrics (such as number of standard blocks processed per second) that highlight different features of the processor to estimate its performance. Example of such programs are AES, RSA, image kernel convolution, ...

Synthetic benchmarks This last category summarizes results from different programs (that might be contained in the second category above) to compute a *benchmark score*. The main advantage of this evaluation method is that it enables comparison between different platforms. However, because all the information is condensed into a single number, it might be hard to evaluate an architecture in detail. Examples of such benchmarks are the MEDIABENCH [Lee 97, Bishop 99] or EEMBC [Levy 05, Kanter 06], both targeted specifically to embedded processors. Often, other benchmarks (such as the SPEC[6] suite) are used to estimate the general-purpose processors present in desktop computers.

Testing and evaluating performance of a processor is a complex process because it involves many components, not only hardware but also software, such as adequate compiler support for the architecture. However, the purpose of this section is not to analyze the processor in depth so it can be compared with every other 32-bit processor available but to give an overview of the capabilities of the processor itself. Even when comparing ULYSSE with other similar processors, precautions have to be taken when interpreting the differences. First, we have to consider that the GCC compiler for ULYSSE does not yet support optimizations, which translates into big and inefficient code. Second, we also have to keep in mind that the main objective we pursued during the development of the processor was flexibility and not performance.

As a result, we propose in this section to evaluate the processor on several test programs that cover different domains of computations spanned by embedded systems. Of course, using a complete benchmark such as *DENBench* that *"[...] allows users to approximate the performance of processor subsystems in multimedia tasks such as image, video, and audio file compression and decompression"*[7] would have provided a better overview of the architecture, but this option was discarded because neither the suite nor its source code are free and because the compiler is still under development. However, some programs similar to test of the EEMBC suite could be written from scratch, the descriptions provided in documents such as [EEM 08] being clear enough to write a test program that captures the original idea and were used for our test suite.

All the test programs were compiled using GCC in `-O0` mode and, when possible, we also compiled the programs for a NXP (founded by Philips) LPC2129 microcontroller[8]. This processor contains an ARM7TDMI-S core from ARM that can be clocked up to 60 MHz. Because the ARM GCC backend allows optimizations, we compiled and measured size and execution of two versions of the ARM program, one with the same compilation options used with ULYSSE and one with optimization flags set to `-O3`, which corresponds to maximum optimization.

The execution times on the ARM platform were measured using the internal high-precision timer that provides a 0.1 µs resolution. A similar technique was used with the ULYSSE processor using the

[6] http://www.spec.org/

[7] Taken from EEMBC website `http://www.eembc.org/benchmark/digital_entertainment_sl.php`

[8] `http://www.nxp.com/acrobat/datasheets/LPC2109_2119_2129_6.pdf`

timer FU, however with a resolution limited to 1 µs. The code template that was used in both cases can be found in the appendix of this document (see section A.7, page 230).

5.5.1 Description of the benchmark suite

We describe in this section the programs that were used to estimate ULYSSE's performance in domains such as multimedia, with video compression or image filtering, or more general purpose, such as sorting. Before that, we need to introduce the notion of *acceleration factor*, which will be used to compare the performance of programs when executed on different platforms.

Definition If we consider a program p, $t_a(p)$ its execution time within CPU_a and $t_b(p)$ its execution time within CPU_b, the *acceleration factor* γ can be considered as:

$$\gamma_a^b(p) = \frac{t_a(p)}{t_b(p)} \tag{5.1}$$

The benchmark suite contains the following programs:

FDCT The forward discrete cosine transform (FDCT) program works on 8x8 image (8 bits deep) blocks. FDCT is a common operation in image compression where it is used to extract the most significant frequency components of an image block.

IDCT The inverse operation of the FDCT that allows the transformation from the frequency domain to the spatial domain.

VLC Variable-length coding (VLC) is a compression scheme used to encode blocks of data, for example coefficients blocks in the JPEG algorithm or inter-block motion vectors in MJPEG. The sample application compresses an array of 128x32 bits of data.

IVLC Inverse variable-length coding (IVLC). This is the inverse function of VLC that transforms compressed data to uncompressed ones. Produces an array of 128x32 bits of data.

High pass greyscale filtering This is a sharpening filter (convolution kernel) that exploits CPU memory accesses and MACC operations. In this sample application, it processes 3x3 gray-scale image blocks.

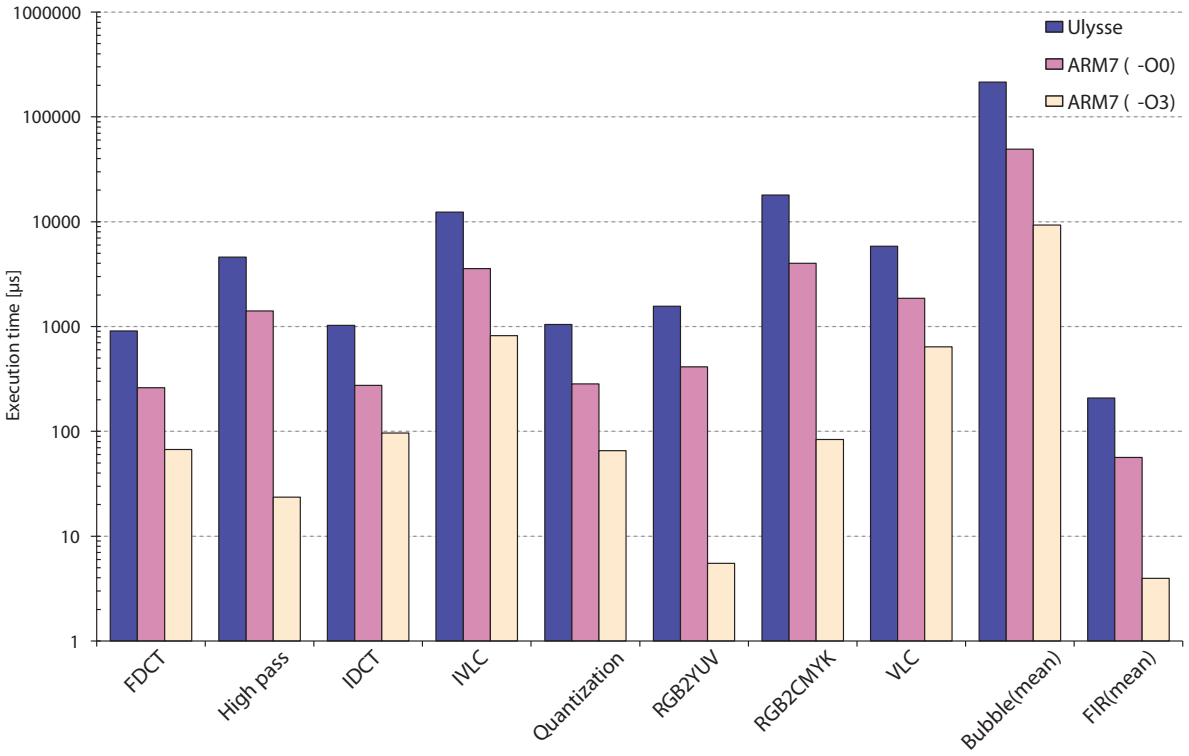
Quantization Digital signal processing technique that reduces the information to be encoded using symbols. Used to perform VLC in image compression for example. The sample application works on a 8x8 data matrix.

RGB to CMYK Color space conversion that consists in transforming Red–Green–Blue (RGB) color coded samples into the Cyan-Magenta-Yellow-blacK (CMYK) colors used in the printing industry or in printers. The arithmetic operations used for this transformation are basic ones such as subtraction along with minima extraction. The image sample is an RGB 100x10 image. The transformation algorithm is based on the EEMBC databook [EEM 08].

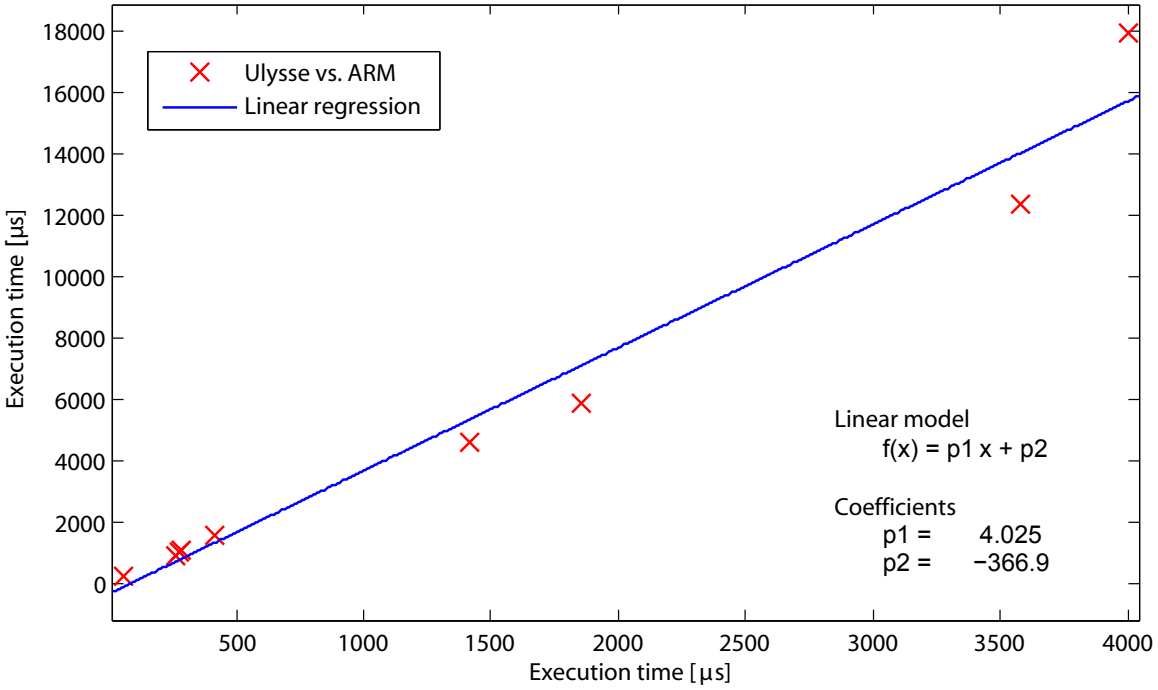
RGB to YUV Color space conversion used in the video reproduction of colors in which RGB samples are transformed in a color space with one luma (Y) and two chrominance components (UV), used for example in the NTSC or PAL video standards. The algorithm's input is an 8x8 RGB sample.

Bubble sort Standard sorting algorithm with complexity $\mathcal{O}(n^2)$, n being the number of elements to sort. In this application sample, an inversely-sorted list of n elements has to be sorted, which constitutes the worst execution time for the algorithm.

FIR Finite Impulse Response filter (FIR) is a type of digital filter that is commonly used in digital signal processing. Its general form is $y[n] = \sum_{i=0}^{N} b_i x[n-i]$ and the $N+1$ elements that compose the right-hand side of the equation are referred as *taps*.



(a) Program performance for each considered architecture



(b) Linear regression of ULYSSE versus ARM7 performance

Figure 5.3: Performance measures on various programs.

Sample program p	Execution time [µs]			Acceleration factor	
	ULYSSE$_a$	ARM7$_b$ (-O0)	ARM7$_c$ (-O3)	$\gamma_a^b(p)$	$\gamma_b^c(p)$
FDCT	910	260.31	67.33	3.50	3.86
High pass	4607	1412.91	23.54	3.26	60.02
IDCT	1026	275.15	96.15	3.73	2.85
IVLC	12359	3579.03	818.82	3.45	4.37
Quantization	1045	283.23	65.36	3.69	4.33
RGB to YUV	1561	413.26	5.5	3.77	75.14
RGB to CMYK	17994	4002.77	83.52	4.49	47.93
VLC	5847	1855.25	638.46	3.15	2.9
Bubble sort [50][a]	10168	2309.5	443.0		
Bubble sort [100]	40836	9243.3	1760.9		
Bubble sort [200]	153445	36985.8	7021.7		
Bubble sort [400]	655338	147964.1	28010.2		
Bubble average	214947	49125.67	9308.95	4.37	5.27
FIR[1]					
FIR[2]	124	33.68	2.33		
FIR[4]	211	56.76	4.00		
FIR[6]	284	76.98	5.66		
FIR[8]	346	94.33	7.33		
FIR average	208	56.56	3.96	3.66	14.26

[a]The parameter corresponds to the *size of the list* to be sorted.

Table 5.1: Performance measures on various programs.

The results of the performance evaluation are summarized in Table 5.1 with more details for the *FIR* and *bubble sort* programs because their performance depend on the number of elements of the working set presented.

In Figure 5.3a, it clearly appears that the performance of the ARM7 platform is better but also that compiler optimizations can be really effective sometimes. Concerning the comparison of the ARM7 and ULYSSE, as can be seen in Table 5.1, for each program p, the $\gamma_a^b(p)$ value remains relatively stable. On average, ULYSSE is 3.7 times slower (with standard deviation $\sigma = 0.43$) than the ARM7TDMI processor. A further analysis of the performance, done with a linear regression[9], depicted in Figure 5.3b, yields a similar estimated ratio (4.025) between the two processors. Even if ULYSSE is outperformed by another CPU, we still consider that this constitutes a reasonable result considering the following:

- The focus was put during the development of the processor on flexibility and correct behavior rather than raw performance.
- The dual-chip, interleaved memory PCB design requires twice the number of cycles another wiring would allow. In other words, using a 32-bit bus instead of 2x16 bits for memory data access would greatly improve the processor performance (at least by 33%).
- The architecture, in and for itself, does not necessarily imply performance as its major advantage. This is especially true when no VLIW instructions are used because a simple operation takes multiple instructions in TTA, hence the slower execution time. Moreover, GCC generated code for ULYSSE does not make use of any architectural optimizations.
- ULYSSE is clocked 20% slower than the particular ARM7 CPU used, notably because it does not benefit from a VLSI implementation as it is the case for the ARM7[10].

[9]Bubble sort and FIR programs were not taken into account for this regression.

[10]A more complete comparison would need to take into account power consumption, silicium size, etc. However, an estimation of ULYSSE's parameters for VLSI was not possible because we did not possess adequate synthesis tools.

5.6 Code size comparisons

Sample program p	Code size [bytes] ULYSSE	ARM7 (-O0)
FDCT	6524	3984
High pass	3580	3060
IDCT	16724	7668
IVLC	17524	5928
Quantization	3848	3240
RGB to YUV	2132	1696
RGB to CMYK	1900	1552
VLC	13816	5752
Bubble sort	5524	4328
FIR	1524	2388

Table 5.2: Code size measurements.

Besides execution time, code size also plays an important role in embedded systems. For this reason, we will now compare the code sizes of the same programs on the ULYSSE and ARM7 platforms[11].

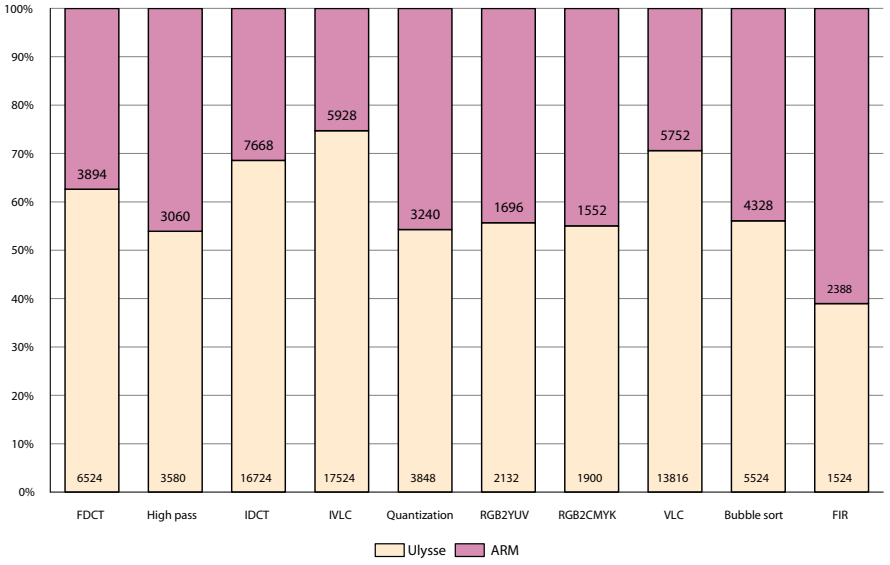


Figure 5.4: Comparison of different program sizes.

Table 5.2 shows the respective code sizes whereas Figure 5.4 depicts a side-by-side comparison of the two compiled codes for each program. On average, ULYSSE code size in our experiments is larger (on average 1.59 times, standard deviation $\sigma = 0.70$) than the ARM version when using the same compiler in the same conditions. Unfortunately, even if it would be interesting to compare the size-optimized versions for both architectures, it is impossible because ULYSSE GCC version does not allow this optimization yet.

It would also be interesting to be able to compare with another processor architecture that tends to overgrow code, such as VLIW, to see how the code increase compares as was done in [Heikkinen 02].

[11] Because the execution times of the ARM7 were collected via a terminal using `printf()` statements, we had to remove them in the size measurements to achieve a fair comparison.

5.7 Comparing simulation and execution times

We have at our disposal two different methods to determine the execution time of a program on the ULYSSE processor, namely *simulation* and *measure*. The advantage of the first method resides in the fact that a real execution platform is not needed to perform measures but also in the fact that the internal signals of the processor can be examined at each execution step thanks to the simulator's display. On the other hand, the second approach overcomes the limitation of the long evaluation times exhibited by the simulator by providing a real-time measure of the execution time.

Sample program	Simulated exec. time [µs]	Execution time [µs]	Simulation time [s]
Quantization	1047.580	1045	95
IDCT	1026.82	1026	92
Bubble[5][a]	97.36	96	5
Bubble[10]	396.56	395	21
Bubble[15]	900.76	899	43
Bubble[20]	1610	1609	83
FIR[1][b]	76.8	75	5
FIR[2]	124.8	124	6
FIR[8]	346.72	346	17

[a] The parameter corresponds to the *size of the list* to be sorted.
[b] The parameter corresponds to the *number of taps* to realize.

Table 5.3: Simulation and execution times.

To demonstrate the validity of the simulation environment measurements, Table 5.3 shows the execution times estimated with the simulator and the measured execution times of some programs. In the same table, an approximate[12] simulation time is shown for each test program, clearly demonstrating that, even if complete simulation remains an interesting solution notably to discover errors in the code, it is a relatively slow process.

5.8 Conclusion

The goal of designing a functional, compiler-supported processor was met. A respectable performance was obtained, as evidenced by the benchmark suite, even if further work on the compiler port to enable the complete spectrum of GCC optimizations would drastically increase the global performance. Another possibility to improve performance would be to use a different memory PCB layout, a change that would directly translate in doubling the performance.

More importantly, the ULYSSE processor provides a stable and well-tested platform that remains widely open to further enhancements. With the robust software support, validated with complex applications, our processor will be used in this thesis for two main purposes:

1. We will show in the next chapter that the flexibility of processor can be further harvested by automatically developing *ad-hoc* functional units to adapt the processor to different programs. Thanks to a genetic algorithm, we will *evolve the processor* in order to accelerate computation in hardware.

2. The ULYSSE processor will be used in the second half of this work as a building block for a massively parallel hardware platform, which will be described starting in chapter 8.

[12] Exact measures are almost impossible, since the simulator does not always return the same measure for the two identical experiments. We estimate the precision to be in the order of ±5 seconds.

Bibliography

[Bishop 99] Benjamin Bishop, Thomas P. Kelliher & Mary Jane Irwin. *A detailed analysis of MediaBench*. In Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'99), pages 448–455, 1999.

[Boule 05] Marc Boule & Zeljko Zilic. *Incorporating Efficient Assertion Checkers into Hardware Emulation*. In Proceedings of the 2005 International Conference on ComputerDesign (ICCD'05), pages 221–228, Washington, USA, 2005. IEEE Computer Society.

[Coppola 04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia & Francesco Papariello. *OCCN: a NoC modeling framework for design exploration*. Journal of Systems Architecture, vol. 50, no. 2-3, pages 129–163, 2004.

[EEM 08] EEMBC. *DENBench 1.0 – Software benchmark databook*, 2008. `http://www.eembc.org/techlit/datasheets/denbench_db.pdf`.

[Eisner 06] Cindy Eisner & Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Foster 03] Harry Foster, David Lacey & Adam Krolnik. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Foster 05] Harry Foster, E. Marschner & D. Shoham. *Introduction to PSL*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Heikkinen 02] J. Heikkinen, J. Takala, A. G. M. Cilio & H. Corporaal. *On efficiency of transport triggered architectures in DSP applications*, pages 25–29. N. Mastorakis, January 2002.

[Kanter 06] David Kanter. *EEMBC Energizes Benchmarking*. Microprocessor Report, vol. 1, pages 1–7, July 2006.

[Lee 97] Chunho Lee, Miodrag Potkonjak & William H. Mangione-Smith. *MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems*. In MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, pages 330–335, Washington, USA, 1997. IEEE Computer Society.

[Levy 05] Markus Levy. *Evaluating Digital Entertainment System Performance*. Computer, vol. 38, no. 7, pages 68–72, 2005.

[Meszaros 03] G. Meszaros, S. M. Smith & J. Andrea. *The Test Automation Manifesto*, vol. 2753 of *Lecture Notes in Computer Science*, pages 73–81. Springer Berlin / Heidelberg, September 2003.

[Pande 05] Partha Pratim Pande, Cristian Grecu, André Ivanov, Resve Saleh & Giovanni de Micheli. *Design, Synthesis, and Test of Network on Chips*. IEEE Design & Test of Computers, vol. 22, no. 5, pages 404–413, 2005.

Chapter 6

Processor evolution

"Le seul mauvais choix est l'absence de choix."

AMÉLIE NOTHOMB, *La métaphysique des tubes*

ULYSSE's main strength resides in its capability to easily accommodate changes in its operation set. As such, it can easily adapt to different kind of computations through its FU-based modular approach. However, in the system described in the previous chapters, the work of determining which FUs are pertinent to realize a given computation is left to the user. In this chapter, we will see that this process of adapting the processor to a particular application can be automated by using a genetic algorithm (GA), a process that not only determines which FUs to include, but also proposes new FUs if necessary.

6.1 Introduction and motivations

As very efficient heuristics, genetic algorithms have been widely used to solve complex optimization problems. However, when the search space to be explored becomes very large, this technique becomes inapplicable or, at least, inefficient. This is the case when GAs are applied to the *partitioning* problem, which is one of the tasks required for the hardware-software codesign of embedded systems.

Consisting in the concurrent realization of the hardware and the software layers of an embedded system, codesign has been used since the early nineties and is now a technique widely spread in the industry. This design methodology permits to exploit the different synergies of hardware and software that can be obtained for a particular embedded system. Such systems are usually built around a core processor that can be connected to hardware modules tailored for a specific application. This "tailoring" corresponds to the codesign of the system and consist in different tasks, as defined in [Harkin 01]: partitioning, co-synthesis, co-verification and co-simulation.

In this chapter, we will focus on the complex, NP-complete [Oudghiri 92], partitioning problem that can be defined as follows: starting from a program to be implemented on a digital system and given a certain execution time and/or size constraints, the partitioning task consists in the determination of which parts of the algorithm have to be implemented in hardware in order to satisfy the given constraints.

As this problem is not new, several methods have been proposed in the past to solve it: Gupta and de Micheli start with a full hardware implementation [Gupta 92], whilst Ernst et al. [Ernst 93, Ernst 02] use profiling results in their COSYMA environment to determine with a simulated annealing algorithm which blocks to move to hardware. Vahid et al. [Vahid 94] use clustering together with a binary-constrained search to minimize hardware size while meeting constraints. Others have proposed approaches such as fuzzy logic [Catania 97], genetic algorithms [Dick 98, Srinivasan 98], hierarchical clustering [Hou 96] or tabu search [Eles 97, Ahmed 04] to solve this task.

We chose to work with a genetic algorithm because of the complex nature of the partitioning task but also because partitioning enables the usage of program *trees* as data structures, a very convenient

data type to apply *genetic programming* (see [Koza 92]). Although some attempts to use genetic algorithms have been shown to be less efficient than other techniques such as simulated annealing to solve the partitioning task [Wiangtong 04, Wiangtong 02], they can be *hybridized* to take into account the particularities of the problem and solve it efficiently.

The improved genetic algorithm we propose starts from a software tree representation and progressively builds a partition of the problem by looking for the best compromise between raw performance and hardware area increase. In other words, it tries to find the most interesting parts of the input program to be implemented in hardware, given a limited amount of resources.

The novelty of our approach lies in the several optimizations passes applied to the intermediary results of a standard GA, which allows to avoid the most common pitfalls associated with GAs, such as being trapped in local minima, thanks to a dynamically-weighted fitness function. Thus, we obtain an hybridized algorithm that explores only the most interesting parts of the solution space and, when good candidates are found, refines them as much as possible to extract their potential. Finally, we will show that our algorithm is robust and performs well on relatively large programs by quickly converging to good solutions.

This chapter is organized as follows: in the next section we formulate the problem in the context of a genetic algorithm and section 6.4 describes the specific enhancements that are applied to the standard GA approach. Afterwards, we present some experimental results which show the efficiency of our approach. Finally, section 6.8 concludes this chapter and introduces future work.

6.2 Partitioning with TTA

We have developed our new partitioning method in the context of the *Move* processor paradigm but our approach remains general and could be used for different processor architectures and various reconfigurable systems with only minor changes. However, the fact that TTAs handle functional units as "black boxes", i.e. without any inherent knowledge of their functionality, implies that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the addresses of the corresponding functional units. As a result, implementing new FUs is simpler than in other architectures.

Because of the versatility of such processors, automatic partitioning becomes indeed very interesting for the synthesis of ontogenetic, application-specific processors: the partitioning can automatically determine which parts of the code of a given program are the best candidates to be implemented as FUs. The algorithm described in this chapter introduces in our global framework a way to *automatically* specializing the instruction set (i.e., defining *ad-hoc* functional units) to the application while keeping the overall structure of the processor unchanged.

6.3 A basic genetic algorithm for partitioning

We describe in this section the basic GA that serves as a basis for our partitioning method and that will be enhanced in section 6.4 with TTA-specific improvements. The basic algorithm, whose flow diagram is depicted in Figure 6.1, works as follows: starting from a program written in a specific language resembling C, a syntactic tree is built and then analyzed by the GA, which then produces a valid, optimized partition. The various parameters of the GA can be specified on the graphical user interface that has been designed, like all other software described in this section, in *Java*.

6.3.1 Programming language and profiling

Assembly could have been used as an input for our algorithm but the general structure of a *Move* assembly program is difficult to capture because every instruction is considered only as a data displacement, introducing a great deal of complexity in the representation of the program's functionality. Thus, the programs to be evolved by the GA are written in a simplified programming language that supports all the classical declarative language constructs in a syntax resembling C. Several limitations have however been imposed to this programming language:

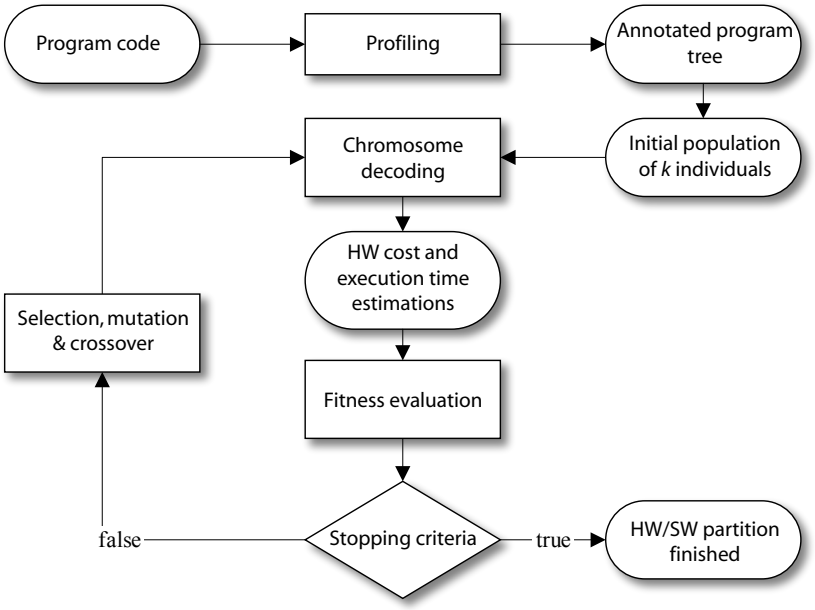


Figure 6.1: General flow diagram of our genetic algorithm.

- Pointers are not supported.
- Recursion is forbidden.
- No typing is considered, all values being treated as 32 bits integers. As a result, only fixed-point or integer calculations can be used.

These simplifications allowed us to focus on the codesign partitioning problem without having to cope with unrelated complications that "real" C programs introduce. However, nothing prevents these limitations to be lifted in an eventual future release of the partitioner.

Prior to being used as an input for the partitioner, the code needs to be *annotated* with code coverage information. To perform this task, we use standard profiling tools on a *Java* equivalent version of the program. We used a *Java* profiler because free and powerful tools exist for this. Even though tools such as *Valgrind*[1] and the *GNU profiler gprof* are available for C, their *Java* equivalent provided a more detailed view of the program execution flow.

This step provides an estimate of how many times each line is executed for a large number of realistic input vectors. With the data obtained, the general program execution scheme can be estimated, which in turn allows the GA to evaluate the most interesting kernels to be moved to hardware.

6.3.2 Genome encoding

Our algorithm starts by analyzing the syntax of the annotated source code. It then generates the corresponding *program tree*, which will then constitute the main data structure the algorithm will work with. From this structure, it builds the *genome* of the program, which consists of an array of boolean values. This array is constructed by associating to each node of the tree a boolean value indicating if the subtree attached to this node is implemented in hardware (Figure 6.2, column a). Since we also want to regroup instructions together to form new FUs, to each statement[2] correspond two additional boolean values that permit the creation of groups of adjacent instructions (Figure 6.2, column b). The first value indicates if a new group has to be created and, in that case, the second value indicates if

[1] `http://www.valgrind.org/`
[2] Statements are assignments, *for*, *while*, *if*, function calls...

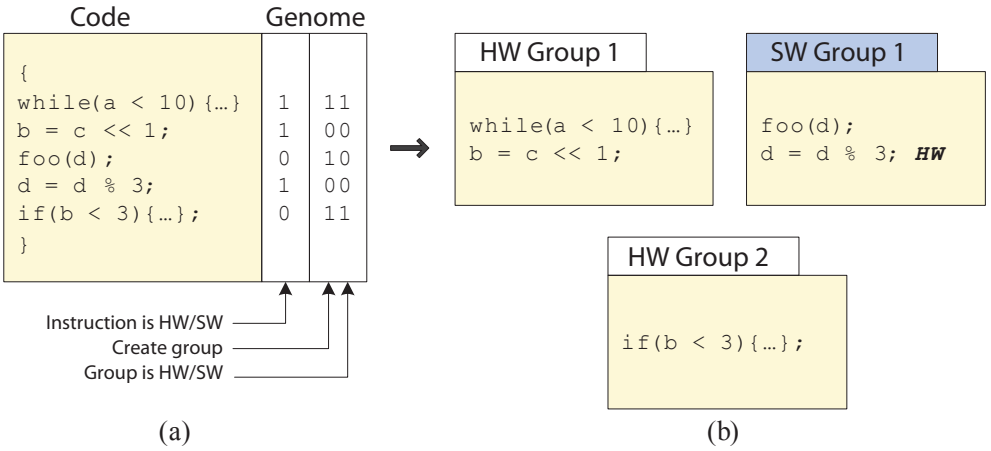


Figure 6.2: Genome encoding.

the whole group has to be implemented in hardware (i.e. to create a new FU). The complete genome of the program is then formed by the concatenation of the genomes of the single nodes.

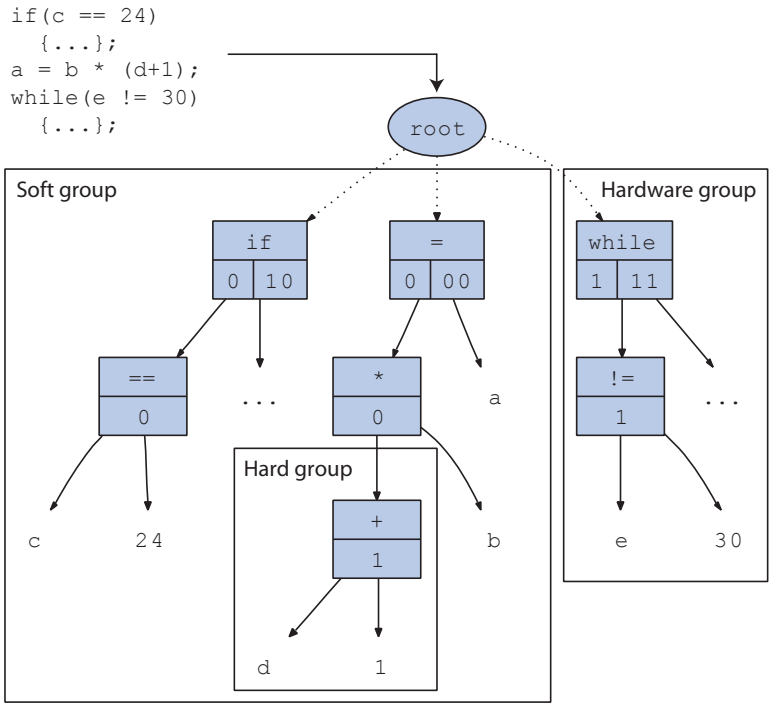


Figure 6.3: Creation of groups according to the genome.

An example of a program tree with its associated genome is represented in Figure 6.3, which depicts the different possible groupings and the representation of the data the algorithm works with.

6.3.3 Genetic operators

When GA are used, several *genetic operators* are applied (see Figure 6.4). In our approach, we use operators inspired from [Koza 92], i.e. selection, mutation and crossover.

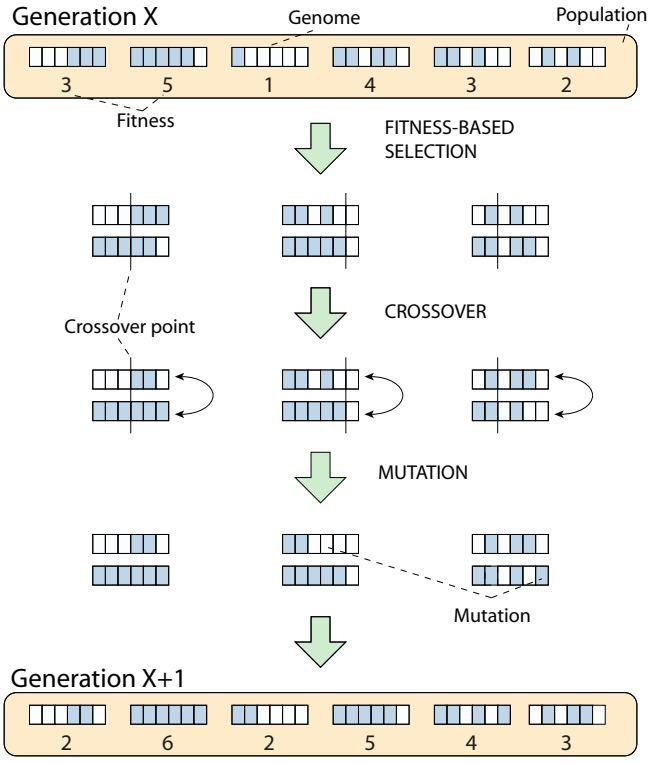


Figure 6.4: Genetic operators.

Selection

The GA starts with a basic population composed of random individuals. For each new generation, individuals are *chosen for reproduction* using rank-based selection with elitism. In order to ensure a larger population diversity, part of the new population is not obtained by reproduction but by random generation, allowing a larger exploration of the search space.

Mutation

A mutation consists in *inverting the binary value of a gene*. However, as a mutation can affect the partitioning differently, depending on where it happens among the genes, different and parameterizable mutation rates are defined for the following cases:

1. A new functional unit is created.
2. An existing functional unit is destroyed. The former hardware group is then implemented in software.
3. A new group of statements is created or two groups are merged together.

Using different mutation rates for the creation and the destruction of functional units can be very useful. For example, increasing the probability of destruction introduces a bias towards fewer FUs.

Crossover

Crossover is applied by randomly choosing a node in each parent's tree and by *exchanging the corresponding sub-trees*. This corresponds to a double-point crossover and it is used to enhance the genetic diversity of the population.

6.3.4 Fitness evaluation

Determining hardware size and execution time

Computing hardware size and execution time is one of the key aspects of the algorithm, as it defines the fitness of an individual which then influences the choice of whether an evolved solution will be kept or not. Different techniques exist to determine these values, for example [Henkel 98] or [Vahid 95] and, more recently, [Pitkänen 05]. The method we chose to use is based on a very fine characterization of each hardware elementary building block of the targeted hardware platform which, in the implementation discussed here, is a Xilinx® VIRTEX™ II-3000 field-programmable gate array.

The characterization of each of these building blocks, which carry out very simple logical and arithmetic operations (`AND`, `OR`, $+$, ...), allows then to arrange them together to elaborate more complex operations that form new FUs in the *Move* processor. For example, it is possible to reduce the execution of several software instructions to only one clock cycle by chaining them in hardware, as depicted in Figure 6.5 (note that the shift operation used in the example is "free" in hardware, i.e. no slices[3] are used, because only wires are required to achieve the same result). This simple example shows the principles of how the basic blocks are chained and how hardware size and execution time are predicted.

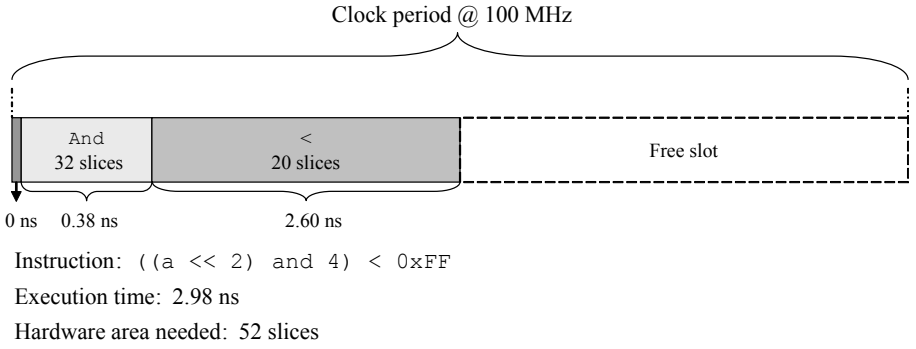


Figure 6.5: Hardware time and size estimation principle of a 32-bit software instruction.

The basic blocks' size and timing metrics have been determined using the SYNPLIFY PRO™ synthesis solution coupled, in some cases, with the Xilinx® place-and-route tools. Thus, we have obtained the number of slices of the FPGA required to implement each block and the length of the critical path of each basic block. Because this characterization mostly depends on the architecture targeted and on the software used, it has to be redone for each different hardware platform targeted. This very detailed characterization permitted us to take into account a wide range of timings, from sub-cycle estimates for combinational operators to multi-cycle estimates for high latency operators such as pipelined dividers. Area estimators were built using the same principles. Using these parameters, determining size and time for each sub-tree is then relatively straightforward because only two different cases have to be considered:

1. For software sub-trees, the estimation is done recursively over the nodes of the program tree, adding at each step the appropriate execution time and potential hardware unit: e.g. the first time an `add` instruction is encountered, an `add` FU must be added to compose the minimal processor necessary to execute this program.

2. For hardware sub-trees, the computation is a bit more complex because it depends on the position of the considered sub-tree: if it is located at the root of a group, it constitutes a new FU and some computation is needed. In fact, the time to move the data to the new FU and the size of the registers required for the storage of the local variables have to be taken into account.

[3]Slices are the fundamental elements of the FPGA. They roughly characterize how much space for logic is available on a given circuit. The name and implementation of these elements differ from one hardware vendor to one another.

Moreover, as every FU is connected to the rest of the processor using a standard bus interface, its cost also has to be considered. Finally, if this unit is used several times, its hardware size has to be counted only once: to determine if the generated FU is new, its sub-trees are compared to the ones belonging the pool of the already available FUs.

A static fitness function

The objective of the GA is to find the partitioning with the smallest execution time whilst remaining smaller than an area constraint. To achieve this, the fitness function we initially used to estimate each individual returned high values for the candidates that balance well the compromise between hardware area and execution speed. Because we made the assumption that the basic solution for the partitioning problem relies on a full software implementation (that is, an implementation based on simple processor that contains the minimum of hardware required to execute the program to be partitioned), we use a *relative fitness function*. This means that this simple processor, whose hardware size is β, has a fitness of 1 and the fitness of the discovered solutions are expressed in terms of a comparison with this trivial solution. We also define α as the time to execute the given program on this minimal processor. For an individual having a size s and requiring a time t to be executed, the following fitness function can then be defined:

$$f(s,t) = \begin{cases} \frac{\alpha}{t} \cdot \frac{\beta}{s} & \text{if } s \leq hwLimit \\ (\log(s - hwLimit) + 1)^{-1} & \text{otherwise} \end{cases}$$

where $hwLimit$ is the maximum hardware size allowed to implement the processor with the new FUs defined by the partitioning algorithm.

The first ratio appearing in the top equation corresponds to the speedup obtained with this individual and the second ratio corresponds to its hardware size increase. Therefore, the following behavior can be achieved: when the speed increase obtained during one step of the evolution is relatively larger than the hardware increase needed to obtain this new performance, the fitness increases. In other words, the hardware investment for obtaining better performance has to be small enough to be retained.

A dynamic fitness function

One drawback of the static fitness function is that it does not necessarily use the entire available hardware. As this property might be desirable, particularly when a given amount of hardware is available and would be lost if not used, we introduce here a dynamically-weighted fitness function that can cope with such situations. In fact, we have seen that the static fitness function increases only when the hardware investment is balanced by a sufficient speedup.

To go further, our idea is to push evolution towards solutions that use more hardware by modifying the balance between hardware size and speedup in the fitness function. This change is applied only when a relatively good solution is found, as we do not want the algorithm to be biased towards solutions with a large hardware cost at the beginning of evolution.

To achieve this goal, a new dynamic parameter is added to the static fitness function and permits more "expensive" blocks to be used when good solutions are found.

For an individual having an hardware size of s, we first compute the adaptive factor k using the following equation:

$$k = \frac{hwLimit - s}{hwLimit}$$

We then compute the individual fitness using that adaptive factor in the refined fitness function:

$$f(s,t) = \begin{cases} \frac{\alpha}{t} \cdot (k \cdot \frac{\beta}{s} - k + 1) & \text{if } s \leq hwLimit \\ (\log(s - hwLimit) + 1)^{-1} & \text{otherwise} \end{cases}$$

where α, β, and $hwLimit$ have the same meaning as in the static function. Thus, we obtain the fitness landscape shown in Figure 6.6, which clearly shows the decrease of the fitness when a given

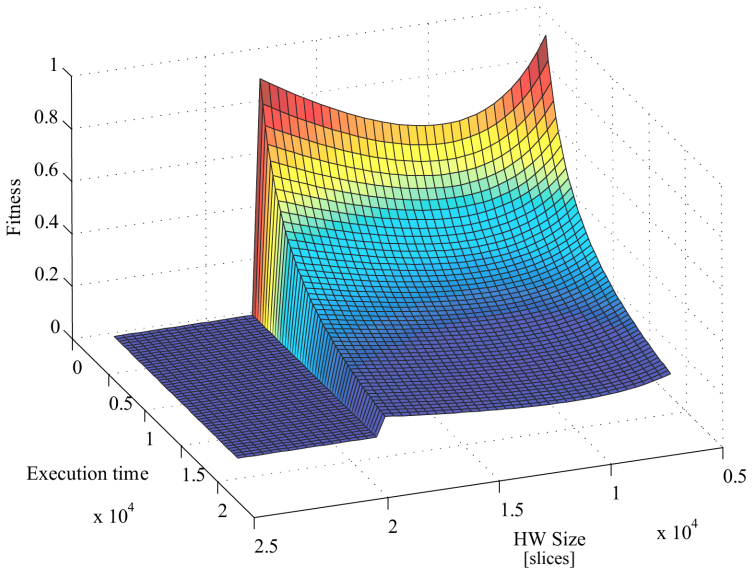


Figure 6.6: Ideal fitness landscape shape for a given program.

$hwLimit$ (on the example given, about 19000) is exceeded. The figure also clearly shows the influence of the k factor which is responsible for the peak appearing near the $hwLimit$.

6.4 An hybrid genetic algorithm

All the approaches described in section 6.1 work at a specific *granularity level*[4] that does not change during the codesign process, that is, these partitioners work well only for certain types of inputs (task graphs for example) but cannot be used in other contexts. However, more recent work [Henkel 01] has introduced techniques that can cope with different granularities during the partitioning. Because of the enormous search space that a real-world application generates, it is difficult for a generic GA such as the one we just presented to be competitive against state-of-the-art partitioning algorithms. However, we will show in the rest of this section that it is possible to *hybridize* (in the sense of [Renders 94]) the GA to considerably improve its performance.

This hybridization translates into the implementation of three additionnal passes during the execution of the algorithm in order to minimize the negative effects that may appear when using genetic algorithms.

6.4.1 Leveling the representation via hierarchical clustering

One problem of the basic GA described above lies in the fact that it implicitly favors the implementation in hardware of nodes close to the root. In fact, when a node is changed to hardware its whole sub-tree is also changed and the genes corresponding to the sub-nodes are no longer affected by the evolutionary process. If this occurs for an individual that has a good fitness, the evolution may stay trapped in a local maximum, because it will never explore the possibility of using smaller functional units within that hardware sub-tree.

The solution we propose resides in the decomposition of the program tree into different levels that correspond to *blocks* in the program[5], as depicted in Figure 6.7. Function calls have the level of the called function's block and a block has level $n + 1$ if the highest level of the block or function calls

[4] Function level, control level, dataflow level, instruction level...
[5] Series of instructions delimited by brackets.

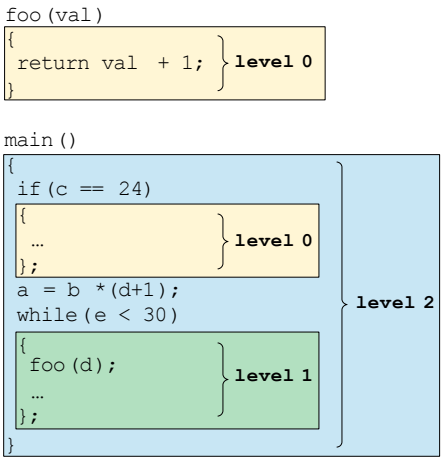


Figure 6.7: Levels definition.

it contains is n, the deepest blocks being at level 0 by definition. These levels represent interesting points of separation because they often correspond to the most computationally intensive parts of the programs (e.g. loops) that are good candidates for being implemented in new FUs.

To circumvent the creation of ill-sized blocks, the GA is recursively applied to each level, starting with the deepest ones ($n = 0$). To pass information between each level, the genome of the best individual evolved at each level is stored. A mutated version of this genome is then used for each new individual created at the next level.

This approach permits to construct the solution progressively by trying to find the optimal solution of each level. It gives priority to nodes close to the leaves to express themselves, and thus good solutions will not be hidden by higher level groups. By examining the problem at different levels we obtain different granularities for the partitioning. As a result, with a single algorithm, we cover levels ranging from instruction level to process level (cf. [Henkel 01] for a definition of these terms). This specific optimization also dramatically reduces the search space of the algorithm as it only has to work on small trees representing different levels of complexity in the program. By doing so, the search time is greatly reduced while preserving the global quality of the solution.

6.4.2 Pattern-matching optimization

A very hard challenge for evolution is to find *reusable* functional units that can be employed at different locations in a program. Two different reasons explain this difficulty, the first being that even if a block could be used elsewhere within the tree, the GA has to find it only by random mutations. The second reason is that it is possible that, although one FU might not be interesting when used once, it would become so when reused several times, because the initial hardware investment has to be made only once.

To help the evolution to find such blocks, a *pattern matching* step has been added: every time a piece of code is transformed in hardware, similar pieces are searched in the whole program tree and mutated to become hardware as well. This situation is depicted in Figure 6.8: starting from an implementation using one FU (Figure 6.8.a), this step searches for candidates sub-trees that show a structure similar to the existing FU. A perfect match is not required: variables values, for example, are passed as parameters to the FU and can differ (Figure 6.8.b). Finally, the software sub-tree is simply replaced by a *call* to that FU (Figure 6.8.c). Reusability is thus greatly improved because only one occurrence of a block has to be found, the others being automatically assigned by this new step.

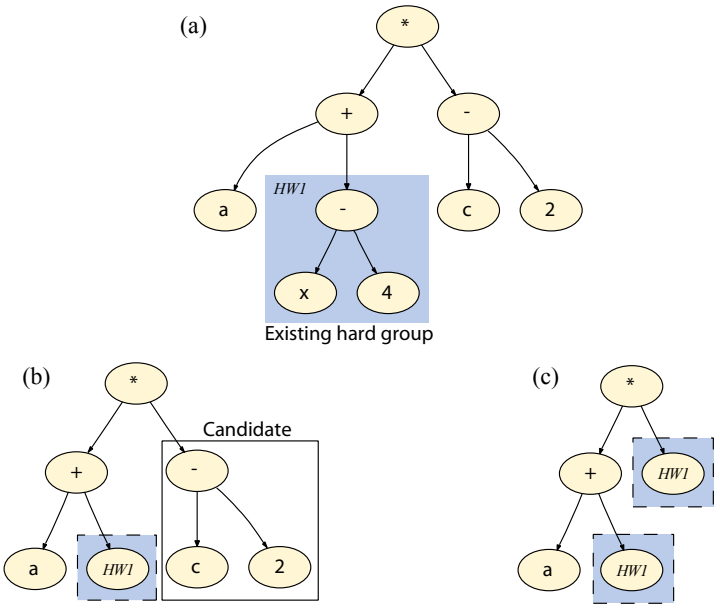


Figure 6.8: Candidates for pattern-matching removal.

6.4.3 Non-optimal block pruning

Another help is given to the algorithm by "cleaning up" the best individual of each generation. This is done by removing all the non-optimal hardware blocks from the genome. These blocks are detected by computing, for each block or group of similar blocks, the fitness of the individual when that part is implemented in software. If the latter is bigger or equal than the original fitness, it means that the considered block does not increase or could even decrease the fitness and is therefore useless. The genome is thus changed so that the part in question is no longer implemented as a functional unit.

This particular step, which could be considered as a *cleaning pass*, was added to remove blocks that were discovered during evolution but that were not useful for the partition.

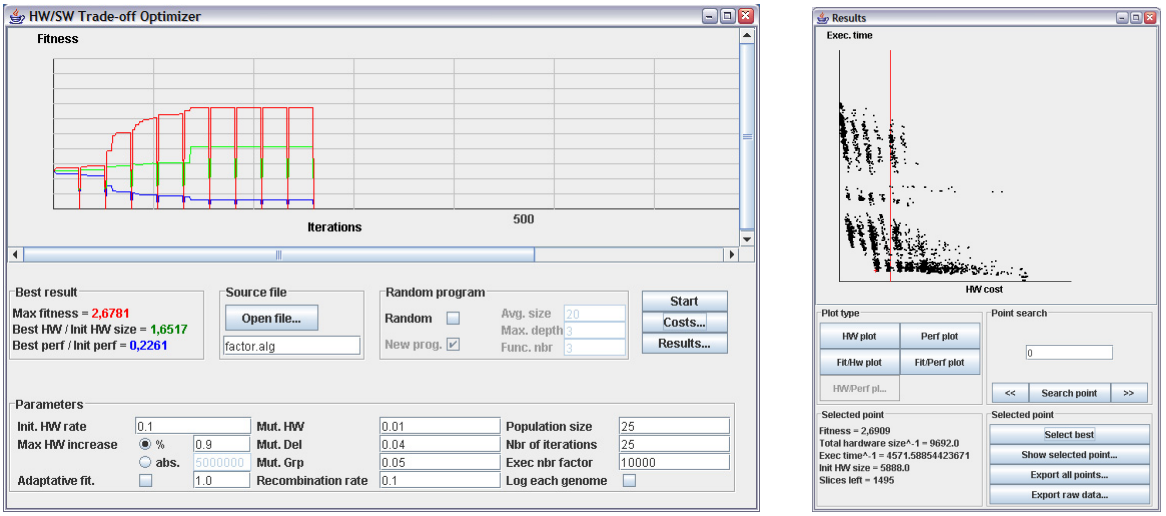
6.5 User interface

The above GA partitions quite efficiently programs of useful size, as we shall see. However, the algorithm itself could obviously be ameliorated by exploiting the latest evolutionary techniques and especially by identifying the correct parameters for the system. To this end, we designed a graphical user interface based on two main windows (Figure 6.9).

The execution window allows the user to set the conventional and the domain-specific parameters of the algorithm. Beside the "standard" mutation and recombination rates, population size, and number of iterations, the key parameters (bottom left) are: the hardware rate of the initial population, the maximum hardware increase with respect to the minimal processor (defined either as a percentage or as an absolute number), and whether an adaptive fitness is used (and, if not, the value of k). In addition, a sub-window (not shown) allows the user to change the hardware costs of the basic blocks used to evaluate the size of the processor.

The middle row of commands in the window determines the program to which the algorithm will be applied (loaded from a source file or generated randomly to allow faster benchmarking) and displays the current results of the evolution as numbers that correspond to the graphical display at the top of the window. These results show the fitness achieved by the best individual in the current population and the hardware and performance increases for this individual.

Once the evolutionary run is completed, the results window can display various representations of the output of the system. Firstly, the performance of the whole evolutionary search can be explored vi-



(a) Execution (b) Results

Figure 6.9: The two main windows of the user interface.

sually using various two-dimensional graphs of the optimization space from different points of view, a feature that allows the user to estimate the impact of the different parameters of the evolutionary process. Another option allows the user to export the data generated for external analysis, plotting, or reuse.

Finally, every single evolved individual can also be analyzed and displayed. The analysis shows the basic parameters of a given solution. The display shows the final code after optimization, showing the discovered FUs in their utilization context. Using this view, it is for example possible to see where the blocks have been used in the code.

6.6 Operation example

Listing 6.2 shows the effects of the algorithm on a reduced set of functions of the `FACT` program (Listing 6.1), which factorizes large integers in prime numbers. The left column shows the original code, annotated by the profiler with the estimated number of times each line is executed. The right-hand column shows the effect of the partitioning algorithm on these same functions.

The main result of the operation is the creation of a set of dedicated FUs, indicated by the `HW` label. A set of functions (`HW(1)` to `HW(4)`) represent code that has been transformed into hardware. These functions usually represent dedicated instructions that are used often in the program and are annotated with the performance increase they allow, their execution time, the size of hardware required for their implementation, and how often they are used within the code (note that, in the figure, these numbers correspond to the entire program and not just to the functions shown).

In this case, the evolutionary algorithm has also decided to transform an entire function (`log2`) in hardware and to build a dedicated FU that implements directly the entire function. This kind of operation is usually applied to small functions that are executed a large numbers of times in the program (as defined by the profiler).

The results of the evolutionary run that produced the partition shown are significant and can be quantified by means of the estimated *speedup* and *hardware increase*. The speedup is computed by comparing the software-only solution to the final partition and the hardware increase represents the number of slices added to the software-only solution to obtain the final partition. Given a maximum hardware increase of 10% (i.e., the final processor had to be at most 10% larger than the minimal processor, and in fact was 8% larger in the individual shown), in this example evolution was able to find a partition that provides an estimated speedup of 227% in the execution time of the program.

```
function isPrime(u) { #28211
  if(u == 2) {#28211
    return true; #5000
  }
  max = bin_sqrt(u); #28211

  if(u % 2 != 0 && u > 2) { #23211
    for(i = 3; i < max; i + 2) { #105795
      if((u % i) == 0) { #95302
        return false; #7719
      }
    }
    return true; #10493
  }
  return false; #4999
}

function log2(u) { #164924
  x = 0; #164924

  while(u != 0) { #1168952
    u = u >> 1; #1004028
    x = x + 1; #1004028
  }
  return x; #164924
}

function isSquare(u) { #94777
  low = 0; #94777
  high = 0; #94777
  x = 0; #94777

  if(u == 1) { #94777
    return true; #12315
  }

  low = (log2(u) >> 1) - 1; #82462
  high = (log2(u) >> 1) + 1; #82462
  low = pow(2, low); #82462
  high = pow(2, high); #82462

  while(true) { #360086
    x = (low + high) >> 1; #360086

    if(low > high) { #360086
      return false; #69115
    }

    if((x * x) < u) { #290971
      low = x + 1; #129104
    }
    else
    {
     if((x*x) > u) { #161867
       high = x - 1; #148520
     }
     else {
       return true; #13347
     }
}}}
```

Listing 6.1: Original code.

```
function isPrime (u) {
   HW(1)
   HW(3)
   if (u % 2 != 0 & u > 2) {
      for (i = 3;i < max;i + 2) (9.5302) {
         if (u % i == 0){
            return 0;
         }
      }
   return 1;
   }
return 0;
}

[HW] [HW] - function log2 (u) {
   x = 0;
   while (u != 0) (100.4028) {
      u = u >> 1;
      x = x + 1;
   }
   return x;
   }

function isSquare (u){
   low = 0;
   high = 0;
   x = 0;
   HW(1)
   low = log2(u) >> 1 - 1;
   high = log2(u) >> 1 + 1;
   low = pow(2, low);
   high = pow(2, high);
   while (1) (36.0086) {
      x = low + high >> 1;
      HW(4)
      if (x * x < u) {
         low = HW(2);
      }
      else {
         if (x * x > u){
            high = x - 1;
         }
         else{
            return 1;
         }
      }
   }
}

/**********************/
// Block HW(1) (Performance gain : 1.0002143413697018)
// (Exec time : 20.0, HW Size : 49.0) — HW Bloc used 3 times
if (v == 0)
{
  return 1;
}

// Block HW(2) (Performance gain : 0.997791210375102)
// (Exec time : 2.0, HW Size : 38.0) — HW Bloc used 4 times
i + 1

// Block HW(3) (Performance gain : 1.824315957433441)
// (Exec time : 44.0, HW Size : 329.0) — HW Bloc used 3 times
max = bin_sqrt(u);

//HW(4) (Performance gain : 1.0013923806061944)
//(Exec time : 81.0, HW Size : 72.0) — HW Bloc used 1 times
if (low > high)
{
   return 0;
}
/**********************/
```

Listing 6.2: Sample result of partitioning.

6.7 Experimental results

To show the efficiency of our partitioning method, we tested it on two benchmark programs and several randomly-generated ones. The size of the applications tested lies between 60 lines of code for the `DCT` program, which is an integer direct cosine transform, and 300 lines of code for the `FACT` program, which factorizes large integer in prime numbers. The last kind of programs tested are random generated programs with different genome sizes. The quality of our results can be quantified by means of the estimated *speedup* and *hardware increase*. The speedup is computed by comparing the software-only solution to the final partition and the hardware increase represents the number of slices in the VIRTEX^{TM} II-3000 that have to be added to the software-only solution to obtain the final partition.

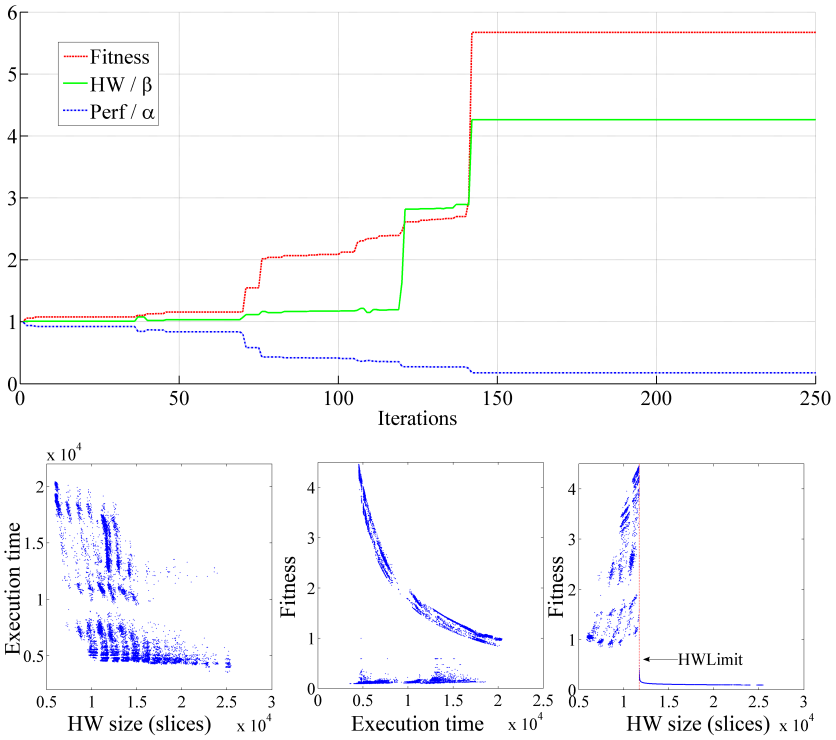


Figure 6.10: Exploration during the evolution (dynamic fitness function).

Figure 6.10 depicts the evolution, using 40 iterations per level, of 30 individuals for the `FACT` program using the dynamic fitness function. A maximum hardware increase of 20% has been specified. We can see that the exploration space is well covered during evolution. Figure 6.11 shows the coverage of the fitness landscape during evolution for both fitness functions along with the best individual trace for the same program.

Table 6.1 sums up the experiments that have been conducted to test our algorithm using both fitness functions. Each figure in the table represents the mean of 500 runs. It is particularly interesting to note that all the results were obtained in the order of a few seconds and not minutes or hours as it is usually the case when GAs are involved. Despite that short running time, the algorithm converged to very efficient solutions during that time.

Unfortunately, even if the domain is a rich source of literature, a direct comparison of our approach to others seems very difficult. Indeed, the large differences that exist in the various design environments and the lack of common benchmarking techniques (which can be explained to the different inputs of HW/SW partitioners) have already been identified in [López-Vallejo 03] to be a major difficulty against direct comparisons.

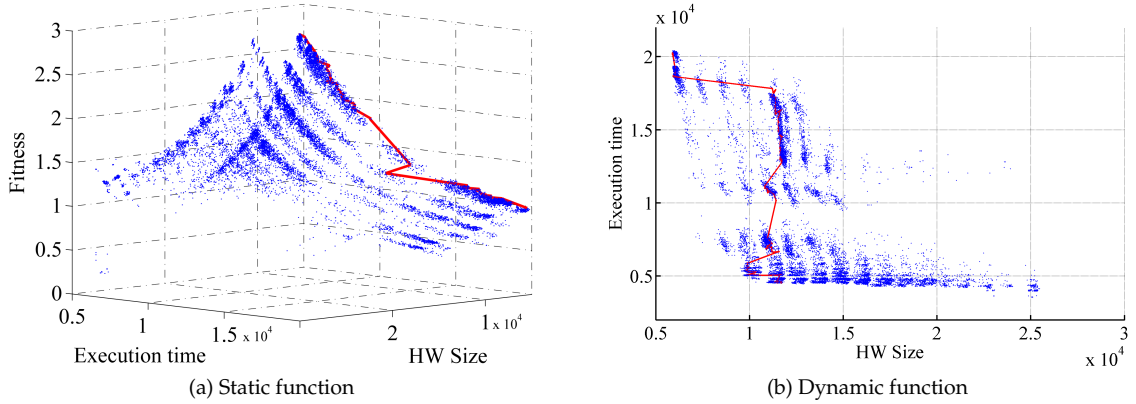
Program name	Gene [bits]	Max HW incr. [slices]	Est. HW. incr. [slices]	Estimated speedup	Run time [ms]
		∞	65.17 %	4.41	3447
FACT	571	20 %	19.75 %	3.00	3750
		10 %	9.66 %	2.38	3864
DCT	212	∞	71.37 %	2.77	547
RND100	100	∞	1.4 %	1.23	250
RND200	200	∞	1.0 %	1.08	734

(a) Static fitness

Program name	Gene [bits]	Max HW incr. [slices]	Est. HW. incr. [slices]	Estimated speedup	Run time [ms]
		∞	213 %	5.69	3250
FACT	571	20 %	19.92 %	2.92	2821
		10 %	9.87 %	1.81	4100
DCT	212	∞	73.73 %	2.77	547
RND100	100	∞	1.4 %	1.23	250
RND200	200	∞	1.0 %	1.08	734

(b) Dynamic fitness

Table 6.1: Evolution results on various programs (mean value of 500 runs).



(a) Static function

(b) Dynamic function

Figure 6.11: Best individual trace along with the explored fitness landscape.

6.8 Conclusion and future work

In this chapter we described an implementation of a new partitioning method using an *hybrid GA* that is able to solve relatively large and constrained problems in a very limited amount of time. When considered in the context of our automated software suite for bio-inspired systems generation, this work can be used to automatically generate *ad hoc* functional units that can be used for the development of ontogenetic processors.

Albeit our method is tailored for the specific kind of processor architecture we are using in this thesis, it remains general and could be used for almost every embedded system architecture with only minor changes.

The use of a dynamically-weighted fitness function introduced additional flexibility in the GA and permitted to closely meet the constraints whilst maintaining an interesting performance. By using several optimization passes, we reduced the search space and made it manageable by a GA. Moreover, the granularity of the partitioning is determined dynamically rather than fixed before execution thanks to hierarchical clustering. The different levels determined by this technique thus represent problems of growing complexity that can be handled incrementally by the algorithm.

The results presented here, as well as those of others groups who have shown that HW/SW partitioning can be successfully used for FPGA soft-cores [Lysecky 05], are encouraging for future research that will address the unresolved issues of our system: for example, although the language in which the problem has to be specified remains simple, the method should be applied to well-known benchmarking suites to validate it more thoroughly. Another idea would be to work not on C code directly but to take advantage of an intermediate representation presenting more regularity, such as GIMPLE (discussed in section 4.2).

Future work within this topic calls for two main axes of research. On one hand it would be interesting to introduce energy as a parameter for the fitness function in order to optimize the power consumption of the desired embedded circuit. On the other hand, it would also be interesting to explore the possibility of automatically generating the HDL code corresponding to the extracted hardware blocks, a tool that would allow us to verify our approach on a larger set of problems and also on real hardware. Another solution would be to use direct C to HDL translation based on languages such as *Handel-C* [Chappell 02] or *Impulse C*[6] [Pellerin 05].

[6] `http://www.impulsec.com`

Bibliography

[Ahmed 04] Usman Ahmed & Gul N. Khan. *Embedded system partitioning with flexible granularity by using a variant of tabu search*. In Proceedings of the Canadian Conference on Electrical and Computer Engineering, vol. 4, pages 2073–2076, May 2004.

[Catania 97] Vincenzo Catania, Michele Malgeri & Marco Russo. *Applying Fuzzy Logic to Codesign Partitioning*. IEEE Micro, vol. 17, no. 3, pages 62–70, 1997.

[Chappell 02] Stephen Chappell & Chris Sullivan. *Handel-C for Co-Processing and Co-Design of Field Programmable System on Chip*. In Proceedings of the Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA02), pages 65–70, 2002.

[Dick 98] Robert P. Dick & Niraj K. Jha. *MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pages 920–935, October 1998.

[Eles 97] Petru Eles, Krysztof Kuchcinski, Zebo Peng & Alexa Doboli. *System Level Hardware/Software Partioning Based on Simulated Annealing and Tabu Search*. Design Automation for Embedded Systems, vol. 2, pages 5–32, 1997.

[Ernst 93] Rolf Ernst, Jörg Henkel & Thomas Benner. *Hardware-Software Cosynthesis for Microcontrollers*. In IEEE Design & Test of Computers, pages 64–75, December 1993.

[Ernst 02] Rolf Ernst, Jörg Henkel & Thomas Benner. *Readings in Hardware/Software Co-Design*, chapter Hardware-software cosynthesis for microcontrollers, pages 18–29. Kluwer Academic Publishers, 2002.

[Gupta 92] Rajesh K. Gupta & Giovanni de Micheli. *System-level Synthesis using Reprogrammable Components*. In Proceedings of the European Design Automation Conference (EDAC), pages 2–7, August 1992.

[Harkin 01] J. Harkin, T. M. McGinnity & L.P. Maguire. *Genetic algorithm driven hardware-software partitioning for dynamically reconfigurable embedded systems*. Microprocessors and Microsystems, vol. 25, no. 5, pages 263–274, August 2001.

[Henkel 98] Jörg Henkel & Rolf Ernst. *High-Level Estimation Techniques for Usage in Hardware/-Software Co-Design*. In Asia and South Pacific Design Automation Conference, pages 353–360, 1998.

[Henkel 01] Jörg Henkel & Rolf Ernst. *An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity that is Driven by High-Level Estimation Techniques*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 2, pages 273–289, April 2001.

[Hou 96] Junwei Hou & Wayne Wolf. *Process Partitioning for Distributed Embedded Systems*. In Proceedings of the 4th International Workshop on Hardware/Software Co-Design (CODES'96), page 70, Washington, USA, 1996. IEEE Computer Society.

[Koza 92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[López-Vallejo 03] Marisa López-Vallejo & Juan Carlos López. *On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques*. ACM Transactions on Design Automation of Electronic Systems, vol. 8, no. 3, July 2003.

[Lysecky 05] Roman Lysecky & Frank Vahid. *A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning*. In Proceedings of the conference on Design, automation and test in Europe (DATE'05), pages 18–23, Washington, USA, 2005. IEEE Computer Society.

[Oudghiri 92] H. Oudghiri & B. Kaminska. *Global weighted scheduling and allocation algorithms*. In Proceedings of the European Conference on Design Automation (DATE'92), pages 491–495, March 1992.

[Pellerin 05] David Pellerin & Edward A. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005.

[Pitkänen 05] Teemu Pitkänen, Tommi Rantanen, Andrea G. M. Cilio & Jarmo Takala. *Hardware Cost Estimation for Application-Specific Processor Design*. In Embedded Computer Systems: Architectures, Modeling, and Simulation, vol. 3553 of *Lecture Notes in Computer Science*, pages 212–221. Springer Berlin / Heidelberg, 2005.

[Renders 94] J.-M. Renders & H. Bersini. *Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways*. In Proceedings of the First IEEE Conference on Evolutionary Computation, vol. 1, pages 312–317, June 1994.

[Srinivasan 98] V. Srinivasan, S. Radhakrishnan & R. Vemuri. *Hardware/software partitioning with integrated hardware design space exploration*. In Proceedings of the conference on Design, automation and test in Europe (DATE'98), pages 28–35, Washington, USA, 1998. IEEE Computer Society.

[Vahid 94] Frank Vahid, Daniel D. Gajski & Jie Gong. *A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning*. In Proceedings of the conference on European design automation (EURO-DAC'94), pages 214–219, 1994.

[Vahid 95] Frank Vahid & Daniel D. Gajski. *Incremental Hardware Estimation During Hardware/-Software Functional Partitioning*. IEEE Transactions on VLSI Systems, vol. 3, no. 3, pages 459–464, September 1995.

[Wiangtong 02] Theerayod Wiangtong, Peter Y.K. Cheung & Wayne Luk. *Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign*. Design Automation for Embedded Systems, vol. 6, no. 4, pages 425–449, July 2002.

[Wiangtong 04] Theerayod Wiangtong. *Hardware/Software Partitioning And Scheduling For Reconfigurable Systems*. PhD thesis, Imperial College London, February 2004.

Part II

Networks of processing elements

Chapter 7

The CONFETTI platform

"The time has come,"
the Walrus said,
"to talk of many things."

LEWIS CARROLL, *The Walrus and The Carpenter*

7.1 Introduction and motivations

A KEY ASPECT of the research that was pursued, first in the Logic Systems Laboratory (LSL) led by Professor Daniel Mange and then in the Cellular Architectures Research Group (CARG), has tried to approach cellular computing by designing large arrays of custom processing elements and by analyzing how some of the mechanisms involved in the development of biological organisms can be effectively applied to these arrays in order to achieve useful properties such as fault tolerance or growth.

Our work represents the continuation of this research, which has traditionally relied on the hardware realization of prototype systems in order to experimentally verify their properties and to analyze their efficiency. Considering the complexity of this kind of systems, their prototyping in hardware requires vast amounts of reconfigurable resources and has led us to the realization of a custom platform specifically designed to implement and test complex cellular computing arrays. In this chapter, we present the salient features of this platform, which we have labeled CONFETTI, for *CONFigurable ElectroniC TIssue*. More precisely, we will present a hardware platform that is able to support the necessary networking and computing elements that can then form what we call a *scalable Network Of Chips*.

The chapter is organized as follows: in the next section, we will give a brief overview of the computational approach that motivated the hardware architecture and describe the state of the art in the domain by analyzing some previous work. The hardware platform itself will then be described in some detail in section 7.3 before discussing several general issues, such as power consumption and integrated test and monitoring, in section 7.4. Finally, section 7.5 concludes this chapter and introduces future work.

Preliminary remark *The large majority of the hardware implementations presented in this chapter – be they schematics or PCBs – was realized by Fabien Vannel in the context of his PhD thesis [Vannel 07], even though we participated in the design and routing of some of the PCBs. On the "middleware side", i.e. the VHDL coding of the FPGA, we reused in this project only the code that Dr. Vannel wrote to drive the RGB screen and reconfigure the cell FPGA from the routing layer (see section 7.3.5). Beside that, every software and hardware component described in this work is our personal contribution.*

7.2 Background

Past research on cellular architectures done by our group, such as [Thoma 04, Rossier 08], has focused on developing a hierarchical approach to design digital hardware that can efficiently implement some specific aspects of this bio-inspired approach. A key aspect of this research is the need for extremely large prototyping platforms involving considerable amounts of programmable logic. This need, along with the non-standard features of our approach, led us to design and build custom platforms that allow us to implement and test in hardware the mechanisms involved.

From a more global perspective, the key question tackled by this research is more or less included in the *evolvable hardware* community, that consists in various groups such as the *Intelligent Systems Group* in York, the *Evolvable Hardware Group* in Brno, the *CRAB* group in Norway or the *Adaptive Systems Research Group* in Japan. The application of bio-inspired mechanisms such as evolution, growth or self-repair in hardware requires resources (fault-detection logic, self-replication mechanisms, ...) that are normally not available in off-the-shelf circuits. For these reasons, over the past several years a number of dedicated hardware devices have been developed, e.g. [Greensted 07, Thoma 04, Upegui 07].

7.2.1 The BioWall

The first platform of this kind done in our group was realized a few years ago thanks to a grant of the Villa Reuge foundation and was destined mainly to illustrate the features of our approach to the general public. The structure of the platform, called the BioWall [Tempesti 01, Tempesti 03] and depicted in Figure 7.1, was centered around the need to clearly display the operation of the system and, as a consequence, it is a very large machine ($5.3m \times 0.6m \times 0.5m$). Intended as a giant reconfigurable computing tissue, the BioWall is composed of 4000 "molecules", each consisting of an 8 by 8 two color LED matrix, one transparent touch sensor and one Spartan® XCS10XL reconfigurable circuit.

Figure 7.1: Photograph of the BioWall.

This "electronic tissue" has been successfully used for prototyping bio-inspired computing machines [Teuscher 03], and has served as a basis for the development of a second bio-inspired architecture, the POEtic tissue [Thoma 04, Thoma 05]. In both cases, the same idea of highly parallel, interconnected cells has served as the background idea for the realization of the architecture.

Despite the fact that the BioWall has fulfilled its role and has been successfully used during several years, it suffers from several limitations which hinder the development of new applications. These limitations are related to the FPGA used, the clock distribution, the autonomy of the system and the display (see for more details [Vannel 07, p. 106]):

- The limitations of the FPGA used in the BioWall are twofold. First, the FPGA model chosen at the time of its conception had to offer good performance at a low price. For these reasons, Spartan XCS10XL were selected. In comparison to what is available on the market today, the performance of these chips is quite limited.

  The second limitation comes from the FPGA configurations. In fact, every FPGA in the BioWall has to use the same configuration, which limits the functionality of every unit to the 10000 equivalent logic gates of the Spartan XCS10XL.

- A a single clock source is used for the whole system. Consequently, the considerable delays inherent in propagating a global signal over distances measured in meters limit the clock speed to about 1 MHz. This fact confines the system to applications where the required computational speed is very low, such as those in which human interaction is required (the intended target of the platform).

- The entire system is controlled by an electronic board connected to a PC and aimed at configuring all the FPGAs and distributing the clock signal to the 4000 FPGAs. Thus, the BioWall acts as a slave electronic system even if the application does not require any interaction with the host computer once configured. This limitation prevents the BioWall from being fully autonomous and introduces a functional bottleneck at the interface between the PC and the reconfigurable logic.

- The huge display of the BioWall is only able to display three different colors because it is composed of red and green LEDs. Furthermore, as the LEDs' intensity can not be changed, there is no brightness control and intermediate shades are not available.

- Each FPGA can communicate only with its direct neighbors and no routing mechanism is present. We will discuss this limitation further in chapter 8.

These drawbacks, along with the evolution of programmable logic devices, have led us to define a novel platform for the implementation of our systems. In the next sections, we will present the structure of the new platform and its salient features.

7.3 The CONFETTI experimentation platform

This novel hardware platform is aimed at the realization of cellular architectures. The system is built hierarchically from a simple computing unit, called the *cell*, which is based on an FPGA and SRAM memory. Several of these units can then be connected, using a high-speed serial communication protocol, to a more complex structure (a stack) consisting of four different kinds of interconnected boards (computational, routing, power supply, and display). These stacks can then be joined together to form an arbitrarily large parallel network of programmable circuits. This structure, while theoretically universal in its operation, is however particularly suited for the implementation of cellular computing, as we will see.

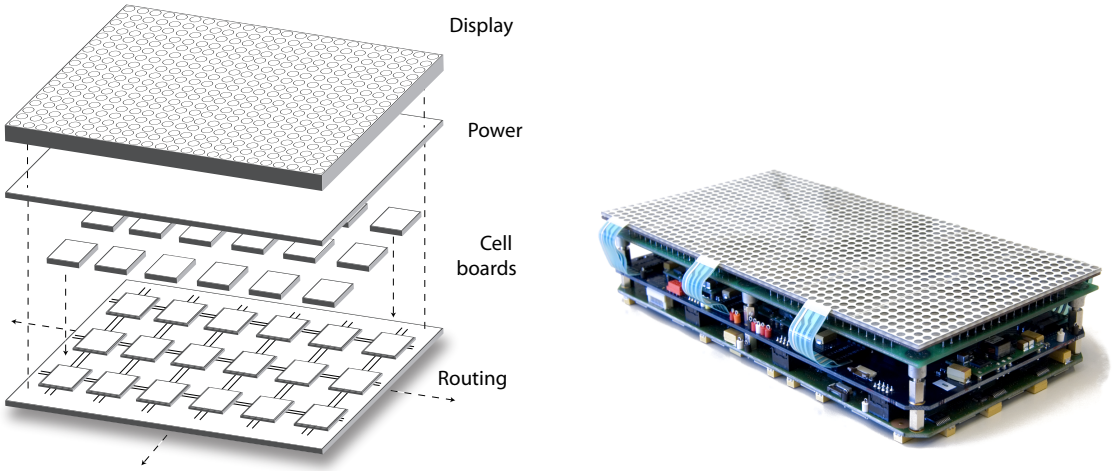
The platform consists of a scalable array of homogeneous processing nodes physically arranged as a networked computing mesh. It is mainly characterized by its flexibility, obtained through a dual-layered arrangement of FPGAs, with a layer dedicated to processing and the other to networking.

This strict separation of computing and networking resources enables independent experimentation of new algorithms in both domains as well as at their interface.

As we will see, the platform tries to avoid the BioWall's shortcomings by proposing an increased amount of versatility and interchangeability in the different constituting elements of the hardware system. Moreover, the system is built hierarchically by connecting elements of increasing complexity which permits to handle more easily the intricacy of the whole system.

7.3.1 Overview – A hierarchical construction

The platform is composed of a set of stacks of printed-circuit boards (PCBs) (Figure 7.2), which can be connected together side by side. Connecting several stacks together potentially allows the creation of an arbitrarily large surface of programmable logic that is used, in this present work, as a network of CPUs where all components can be reconfigured.



(a) Schematic (b) Photograph

Figure 7.2: A stack schematic and photograph.

Each of these stacks is composed of four kinds of boards:

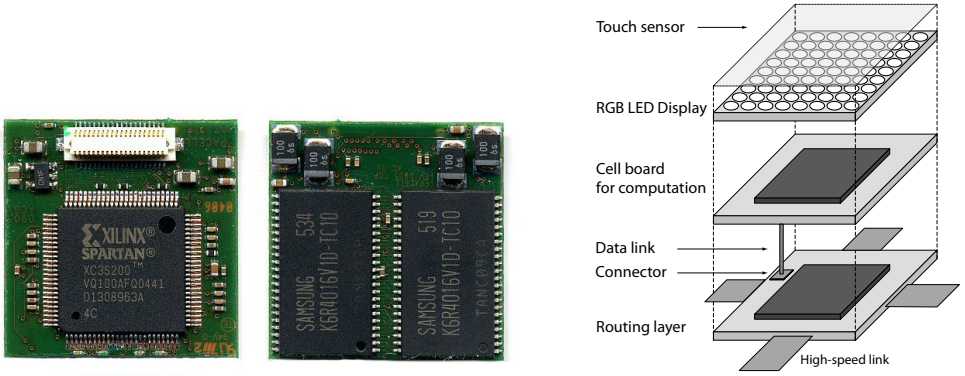
1. **The cell boards**
   Up to 18 of these boards can be used per stack. They represent the computational part of the system and are composed of an FPGA and an 8 Mbit static memory. Each board is directly plugged, via a connector, into a corresponding routing FPGA in the subjacent routing board.

2. **The routing board**
   This board constitutes the communication layer of the system. Articulated around 18 FPGAs in a mesh topology, the board can implement different routing algorithms to form a network that provides inter-FPGA communication but also communication to other routing boards.

3. **The display board**
   The topmost layer of the stack is composed of a display board that consists of a RGB matrix LED display to which a touch sensitive surface has been added.

4. **Power supply board**
   Above the routing layer lies a board that generates all the power supplies required by the system and handles functions such as startup and monitoring.

This hardware structure proposes an increased amount of versatility compared to other platforms, notably because its modular organization allows interchanging all the elements of the system. If this might be interesting in the perspective of debugging and replacing faulty parts, the clear advantage of this approach resides in the fact that the computing elements, which are plugged into the system and not soldered on it, could also be easily replaced. This latter option is of particular interest in the larger perspective of a prototyping board for unconventional computing: nothing prevents the replacement of the current cells with more “exotic” or non-standard units that could potentially be of interest for research in bio-inspired mechanisms.



(a) SRAM chips, FPGA and connector to routing layer

(b) Cell board schematic overview

Figure 7.3: Pictures and schematic of the cell board.

7.3.2 The cell board

This small PCB, depicted in Figure 7.3a, constitutes the basic building block of our hardware platform. It is articulated around a SPARTAN® 3 FPGA[1] from Xilinx® coupled with 8 Mbit of 10 ns SRAM memory and a temperature measurement chip. Equivalent to 200'000 logic gates, the core of this board possesses some interesting features such as hardware multipliers, 18 kbit of internal dual-port memory and four digital clock managers (DCM) that allow to obtain, from the 50 MHz local clock, working frequencies up to 300 MHz. All these components are soldered on a very small (26×26 mm) eight-layer PCB.

The board contains various connections to the others components of the system that all pass through the connector visible on top of Figure 7.3a and Figure 7.3b. The connectivity of the board is as follows:

Type	Number of lines	Description
High-speed bus	6 LVDS	Differential high-speed communication with the subjacent FPGA on the routing level (3 pairs in each direction), 500 Mbit max. per pair.
Low-speed bus	6	Communication bus to the routing FPGA that carries the display and control signals.
Configuration	5	Used to configure the FPGA via the routing board. The bitstream is sent serially.
Clock	1	50 MHz clock shared with the underlying routing FPGA.
Reset	1	Reset signal generated by the underlying routing FPGA.
Temperature	3	Digital temperature readout.
Power	12	Carries the power supply of the FPGA and SRAM modules.

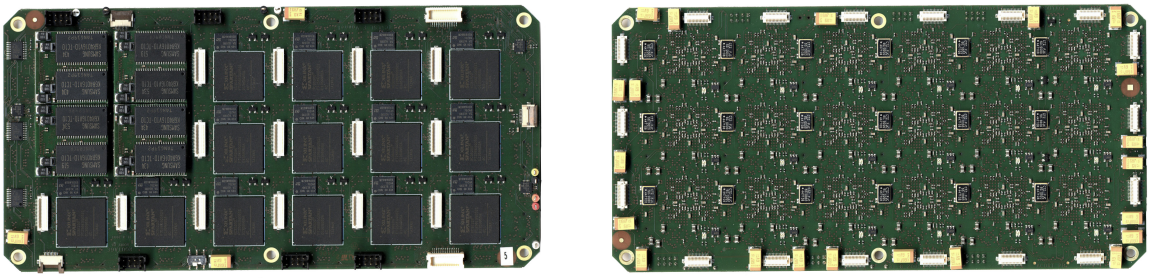
Table 7.1: Cell board I/O signals.

The FPGA allows any soft-core CPU to be instantiated as a computing node. In this work however, we will focus only on the use of the ULYSSE processor. In the context of this platform, the customization ability of the processor will help to develop an ad-hoc network interface with DMA with the capability to modify the processor's code at run-time using specially-formed data packets (see section 8.4.1).

[1]The exact model is the XC3S200 FPGA.

As seen from this board, ULYSSE has access to the touch sensor area above it and to an 8 by 8 slice of the RGB display (Figure 7.3b). The information is passed not via a direct connections to these elements but thanks to a low speed communication bus with the routing board, which is described in the next section.

7.3.3 The routing board



(a) Top view – The mesh grid arrangement of the FPGAs is clearly visible. Four cells have been inserted in top-left corner of the board.

(b) Bottom view – The 50 MHz oscillators are visible along with the inter-board connectors.

Figure 7.4: Routing board pictures.

One of the main challenges in today's hardware architectures resides in implementing versatile communication capabilities that are able to provide a sufficient bandwidth whilst remaining cost- and size-efficient, as evidenced in research on NoC [Bjerregaard 06] and other systems [Moraes 04]. This aspect of parallel systems is unfortunately often ignored in bio-inspired hardware approaches and is another factor that prevent the implementation of complex applications. In our case, the addition of a network layer enables data to be moved not just within the various FUs of the processor but also outside of it to realize a complex *Network Of Chips* of virtually unlimited size.

To accomplish this, the functions of computation and communication were logically and physically split. With this separation, we were able to preserve the whole of the computational resources of the cell board while retaining full flexibility for the communication network in our system within the routing board, which also handles tasks such as system configuration and display management. As it constitutes a subject in itself, *routing and communications* will be discussed in the next chapter (section 8.3, page 121) and we will only describe here the hardware structure of the routing board.

Measuring 192mm× 96mm, this highly complex board (twelve layers) has components soldered on both sides and can host as many as eighteen cell boards. Based on a six-by-three regular grid topology (Figure 7.5), this board is composed of a set of 18 reconfigurable FPGA circuits and 18 Flash memories.

Connectivity

Each of these routing FPGAs is connected to its neighbors with high-speed low-voltage differential signalling (LVDS) lines. As the routing boards constitute the communicating backplane of the whole CONFETTI, connections between the different stacks are also implemented here. External connectors are present on the four sides of the board and provide the same connectivity as the links between the FPGAs: two adjacent routing boards then represent effectively a single uniform surface of FPGAs. The modularity of the setup allows the creation of systems consisting of several stacks that behave as a single, larger stack.

Configuration of the FPGAs

In parallel to these very high speed links that are used for communication, another bus, working at 50 MHz, is also present on the routing layer. This bus, as we will see now, is used for transmitting,

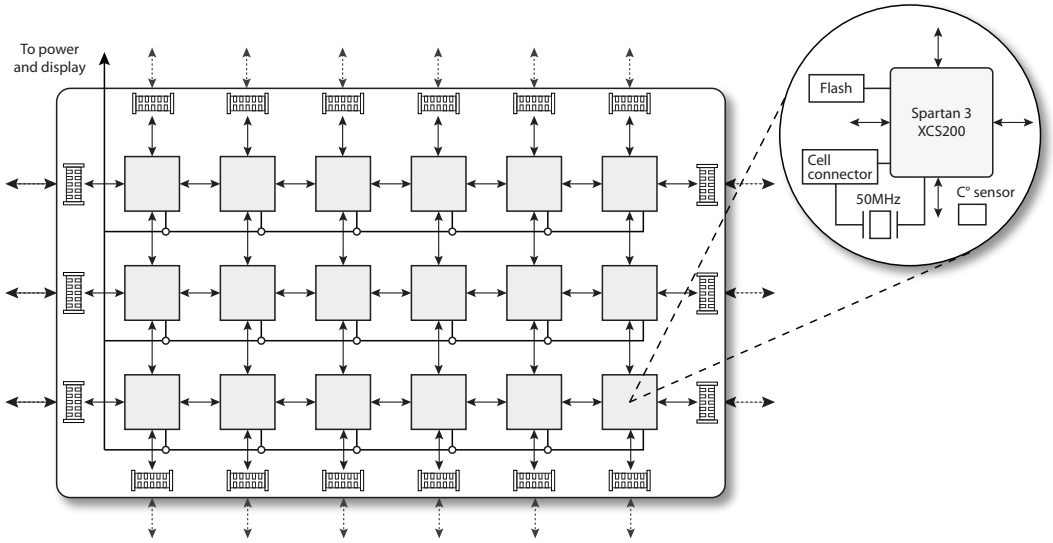


Figure 7.5: Routing board schematic.

loading and storing different cell configurations.

The various reconfigurable circuits used in CONFETTI are all based on SRAM technology, which means they can be reconfigured an unlimited amount of times and that the configuration requires a relatively short time (typically 20 ms). The routing FPGAs all share the same configuration bitstream, stored in a memory which can be accessed via a JTAG connector on the routing board, i.e. all the routing FPGAs implement the same circuit. The situation is more complex for the FPGAs on the cell boards: as they could have a different configuration and even be reconfigured dynamically, one of the problems that need to be addressed by the system is how to convey the correct configuration to each of the cell FPGAs.

This task is performed within the routing board: each routing FPGA can access an adjacent 16 Mbit Flash memory, which can be used to store as many as sixteen different configurations for the cell FPGAs or serve as non-volatile memory available for applications.

The contents of this memory can be modified using an external interface connected to a computer (more on this in section 7.4.3). At the moment, this constitutes the only way to store new applications bitstreams to be executed by the cells. Of course, the configurability of the routing FPGAs enables almost unlimited versatility in the configuration scheme, allowing for example the implementation of applications that would update the Flash contents using external memories or retrieve the cell configurations from sources other than the local Flash memory, such as LAN or WiFi connections.

7.3.4 The power supply board

The SPARTAN® 3 FPGAs that are used throughout the CONFETTI system are built on a 90 nm CMOS process. This gives them the advantage of being very fast and of having lots of embedded features but also has the disadvantage of needing several low-power voltages. Thus, the FPGA core is powered by a 1.2 V voltage but also needs 2.5 V for the LVDS interface and for configuration purposes. Finally, a 3.3 V voltage is needed to interface with the Flash and SRAM memories. Moreover, all these voltages need to be well stabilized, as Xilinx® FPGAs allow only ±5 % tolerance.

To cope with all these requirements and the fact that the eighteen cells and the routing board are not only very complex but also power-hungry, a dedicated power supply board was added. This six-layer PCB, mainly responsible of supplying the correct voltage to all the components on the boards underneath, has the same size as the routing board. Articulated around six DC / DC converter, this board generates from a global 5 V the three mentioned voltages of 1.2 V, 2.5 V and 3.3 V that are then brought, using six 8-pins connectors, to the routing board where they are used to power-up the

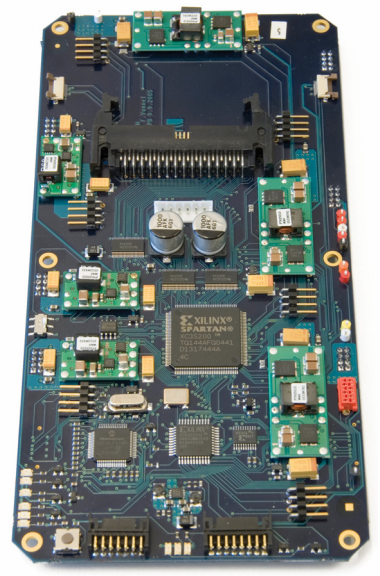


Figure 7.6: A power supply board.

different FPGAs and processors present.

Due to the high complexity of all the boards described, a micro-controller on the power supply board also acts as a *supervisor* and checks several factors, such as the stability of the power supplies and the power consumption, with the aim of preventing failures. This micro-controller is also responsible for supervising the start-up of the whole system, a rather complex sequence that involves switching on the DC / DC converters, controlling the stability of all voltages, configuring the routing FPGAs, and monitoring the temperatures of all PCBs. If any of these tests should fail, the entire system is switched off to prevent damages.

7.3.5 The display board

Unlike the above-mentioned BioWall, which was primarily a demonstrator, the main purpose of CONFETTI is the high-speed prototyping of complex multi-cell systems. Nevertheless, the success of the earlier machine led us to integrate in the new one a relatively simple display. On the very top of the stack lies then a 24-bit RGB LED display capable of displaying 48×24 pixels that can be refreshed at a rate of 100 times per second.

Because the display only includes a reduced framebuffer, a fast external component is required to ensure a smooth display. This task has been given to a dedicated SPARTAN® 3, physically located on the power supply board, whose task is to manage a whole framebuffer and to collect from each cell the data to be displayed.

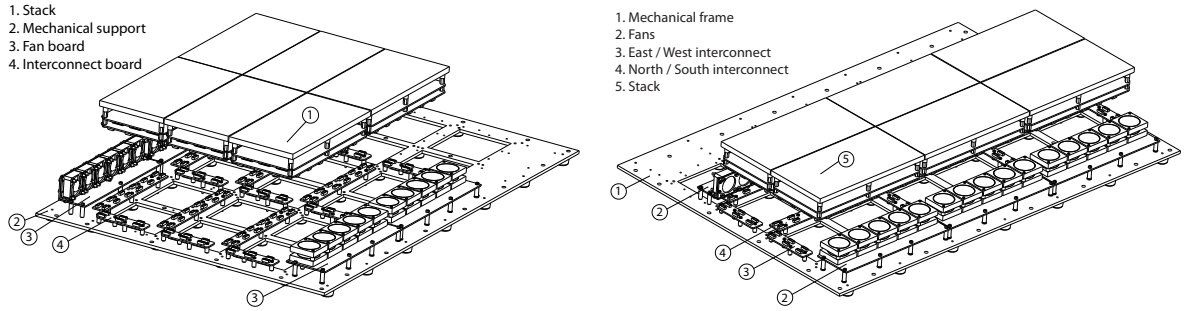
The purpose of this display is to provide a distributed overview of the operation of the system, for example to illustrate its operation or to display patterns such as network congestion or thermal buildup. It is worth noting here that each cell has access to only part of the screen, namely a square of eight by eight pixels directly above it. To provide a direct human interface to the system, a touch-sensitive surface was glued to each square.

Even if the resolution available for each cell is very limited, the main advantage of this kind of screen resides in the fact that it is possible to put several screens border to border without any gap, a necessary feature in view of building large systems consisting of several stacks side by side.

7.4 The CONFETTI system

The previous section has been devoted to the description of a complete basic element: the stack. As we mentioned earlier, a complete CONFETTI system consists of an arbitrary number of such stacks seamlessly joined together (through the border connectors in the routing board) in a two-dimensional array.

The current test configuration that has been built and tested, for example, consists of six stacks in a 3 by 2 array, as shown in Figure 7.7. The same figure also depicts a different layout of the boards where a 2 by 3 array is used. One can also note the presence of a metallic frame to provide a rigid structure that ensures good connections between the different stacks.



(a) Layout 1 (b) Layout 2

Figure 7.7: The CONFETTI system in two different configurations.

The connection of several boards together potentially allows the creation of arbitrarily large surfaces of programmable logic. However, considerations of power consumption, thermal management, and system monitoring come into play for a system of this kind.

7.4.1 Power consumption considerations

The power consumption of the system is very difficult to determine in advance with a reconfigurable system, since it heavily depends on the FPGA configuration, and will change with each application. However, Xilinx® XPower®[2] tools give a worst-case current consumption of 1 A per core. This estimation is based on XC3S200-VT100-4 FPGA with a clock frequency of 100 MHz and full utilization of the FPGA resources.

As we have a total of thirty-six of these FPGAs for each stack, the maximum current that might be drained could be as high as 36 A on the 1.2 V power supply. In comparison, the power estimation for 2.5 V and 3.3 V are very low, with only 4 A for each. In addition to the FPGAs, the last significant energy consumer is constituted by screen that can use a maximum of 30 W. Thus, the total maximum consumption for one stack is 100 W, provided by an external 5 V regulated power supply to the power board.

7.4.2 Thermal management

Despite the fact that the reconfigurable circuits in the platform use a state-of-the-art fabrication technology that makes them less power hungry than previous generation FPGAs, the number of circuits involved in the whole system makes thermal management a real issue, each stack consuming a maximum total of 100 W. To solve it, we had to implement an adequate cooling system in order to evacuate the generated heat. Thus, we integrated boards with fans on the top and bottom borders of the whole

[2] See [Xil 07] and `http://www.xilinx.com/products/design_resources/power_central/index.htm`

system. Two different types of fan boards have been used: on the bottom, fans are used for introducing cooler air on the FPGAs and on the top, fans extract the hot air away from the boards. Each board contains between six and height fans, each independently controllable.

To ensure a good thermal protection without having to turn on all the available fans, the temperature of several components is constantly monitored on:

- every cell;
- every FPGA on the routing board but also three additional locations on the board;
- every power supply board, with six temperature sensors.

It is then relatively easy to detect thermal hot spots and turn on the necessary fans, a process which is done by a dedicated component described in the next section.

7.4.3 Integrated test and monitoring

Because of the relatively high price of the components and the low number of stacks produced, we took precautions to minimize the risks of component failure due to short circuits or thermal stress.

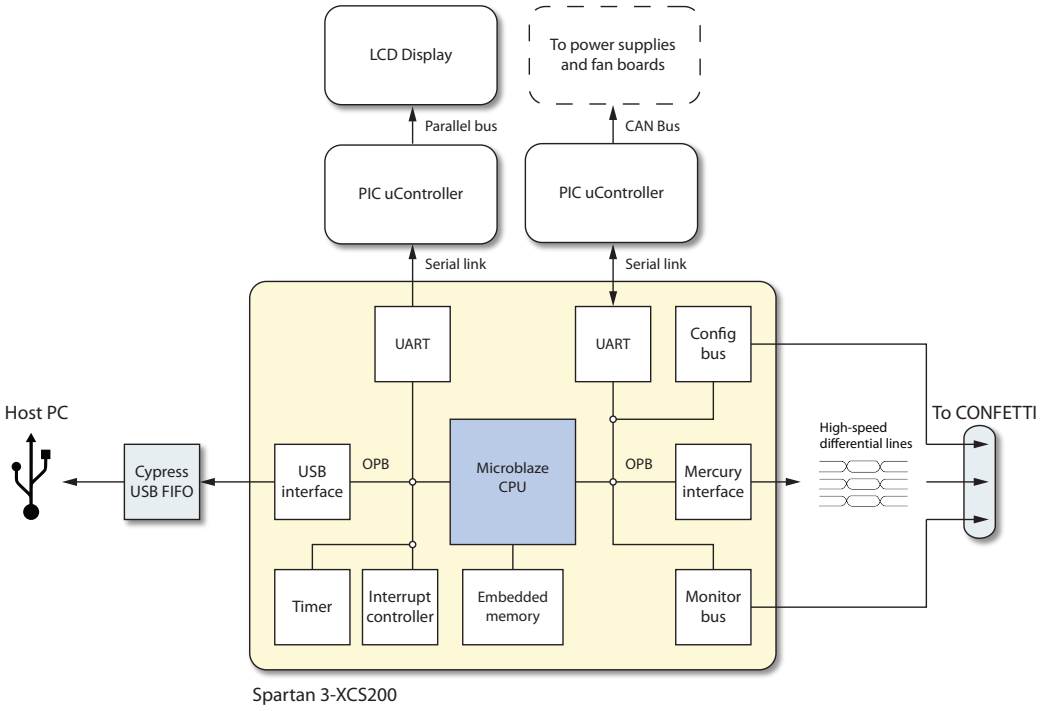


Figure 7.8: Schematic of the monitor and control board.

Thus, each power supply works in limited current mode, in which 20 A at 5 V can be sourced. Moreover, despite the fact that the whole system can be used in a configuration where it is completely independent of any external control mechanism, a supervisor board has been developed in order to monitor the whole system but also to help debugging (Figure 7.8). This board, which lies next to the CONFETTI system, contains several elements:

- a SPARTAN® 3 FPGA in which a MICROBLAZE™ CPU has been instantiated;
- an USB interface chip for the MICROBLAZE™ that allows the transmission of configurations and data;

- a CAN protocol controller, managed by another micro-controller, which handles the different power supplies and fans of the system.

Thanks to the USB connection with an host PC, it becomes possible to monitor and control various parameters related to the support of the CONFETTI platform. Figure 7.9 shows a control software tool that was developed for this purpose and that allows to monitor and switch each power supply and fan individually and globally, to monitor the various temperature sensors, and to control different electrical parameters.

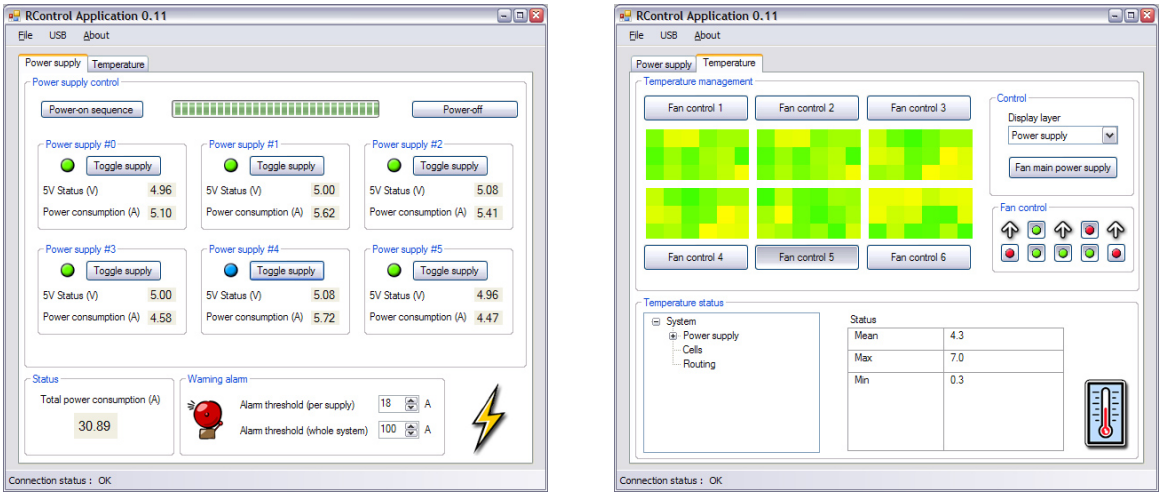


Figure 7.9: Remote control software of the monitoring tools.

7.5 Conclusion and future work

In this chapter, we described a novel hardware platform aimed at the realization of cellular computing applications. The versatility of the platform along with its potential computational power offer very interesting perspectives not only for the exploration of various routing algorithms but also in terms of future developments.

For example, should more processing power be needed, the cell boards could easily be replaced by a bigger reconfigurable device or by a different kind of circuit. Similarly, the system's modularity implies that few changes would be required, for example, to allow different types of cell boards on the same routing substrate.

Aside from the computational aspect, the system is also open to several improvements related to I/O. For example, it is clear that the current external USB connection is insufficient for high bandwidth applications and an improvement to the system is the introduction of high-speed I/O boards that, placed on the borders of the array, could allow the implementation of data-intensive applications (video streaming, for example).

To exploit the features of the system, we will describe in the next chapter the necessary set of tools for routing data packets inside the CONFETTI platform. This done, we will then demonstrate how parallel applications can be implemented easily thanks to different software abstractions organized in layers.

Bibliography

[Bjerregaard 06] Tobias Bjerregaard & Shankar Mahadevan. *A survey of research and practices of Network-on-chip*. ACM Computing Survey, vol. 38, no. 1, page 1, 2006.

[Greensted 07] A.J. Greensted & A.M. Tyrrell. *RISA: A Hardware Platform for Evolutionary Design*. In Proc. IEEE Workshop on Evolvable and Adaptive Hardware (WEAH07), pages 1–7, Honolulu, Hawaii, April 2007.

[Moraes 04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller & Luciano Ost. *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. Integrated VLSI Journal, vol. 38, no. 1, pages 69–93, 2004.

[Rossier 08] Joël Rossier. *Self-Replication of Complex Digital Circuits in Programmable Logic Devices*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2008.

[Tempesti 01] Gianluca Tempesti, Daniel Mange, André Stauffer & Christof Teuscher. *The BioWall: an electronic tissue for prototyping bio-inspired systems*. In Proceedings of the 3rd Nasa/-DoD Workshop on Evolvable Hardware, pages 185–192, Long Beach, California, July 2001. IEEE Computer Society.

[Tempesti 03] Gianluca Tempesti & Christof Teuscher. *Biology Goes Digital: An array of 5,700 Spartan FPGAs brings the BioWall to "life"*. XCell Journal, pages 40–45, Fall 2003.

[Teuscher 03] Christof Teuscher, Daniel Mange, André Stauffer & Gianluca Tempesti. *Bio-Inspired Computing Tissues: Towards Machines that Evolve, Grow, and Learn*. BioSystems, vol. 68, no. 2–3, pages 235–244, February–March 2003.

[Thoma 04] Yann Thoma, Gianluca Tempesti, Eduardo Sanchez & J.-M. Moreno Arostegui. *POEtic: An Electronic Tissue for Bio-Inspired Cellular Applications*. BioSystems, vol. 74, pages 191–200, August 2004.

[Thoma 05] Yann Thoma. *Tissu Numérique Cellulaire à Routage et Configuration Dynamiques*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[Upegui 07] Andres Upegui, Yann Thoma, Eduardo Sanchez, Andres Perez-Uribe, Juan-Manuel Moreno Arostegui & Jordi Madrenas. *The Perplexus bio-inspired reconfigurable circuit*. In Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS07), pages 600–605, Washington, USA, 2007.

[Vannel 07] Fabien Vannel. *Bio-inspired cellular machines: towards a new electronic paper architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, 2007.

[Xil 07] Xilinx. *Xilinx Power Estimator User Guide*, 2007. `http://www.xilinx.com/products/design_resources/power_central/ug440.pdf`.

Chapter 8

CONFETTI communication and software support

"Im Gebirge ist der nächste Weg von Gipfel zu Gipfel:
aber dazu musst du lange Beine haben."

"In the mountains the shortest route is from peak to peak,
but for that you must have long legs."

FRIEDRICH NIETZSCHE, *Also sprach Zarathustra*

WHEN WE described in the previous chapter the shortcomings of the BioWall, we deliberately did not expand on what, in our opinion, constitutes the strongest limitation of the system: the lack of adequate *software support* to program the platform and the impossibility to communicate farther than from the direct neighbor, which greatly limits the range of possible computations.

In this chapter, we will describe how we managed to remove both these constraints in CONFETTI by first implementing an efficient routing algorithm that provides powerful communication capabilities to the cells and then by developing a set of tools that attempt to tackle the software problem by providing a simple but complete software framework to handle the complexity of the system.

8.1 Introduction and motivations

If CONFETTI makes available to the user an impressive amount of raw processing power, the major difficulty of how to harvest this power remains unsolved. Considering that computing generally requires *data* to work on, this implies that without adequate and efficient means to bring these data to the processing nodes, the risk of underusing the resources due to an ill-designed communication system is high. Moreover, using ULYSSE as a processing element implies the opportunity to consider networking as a "natural extension" of the *Move* model in which the data are not displaced only inside a processor but also *outside* of it, extending the paradigm a step further.

With these elements in mind, we will first consider in this chapter the different possibilities offered for communicating. In the next section, we will first detail how local high-speed communication is achieved with the hardware implementation of physical links that provide inter-FPGA communication. We will then explain how these short-distance links can be used in conjunction with a routing element to provide long-distance point-to-point communication between the PEs. After that, section 8.5 will be devoted to the software messaging layer that is built atop that hardware to provide efficient primitives – such as broadcasting – that simplify the programming of networked applications but also acts as an extensible framework for distributed applications. With this software layer, it becomes possible to quickly evaluate new algorithms, techniques and ideas in the field of bio-inspired computing but also to harvest more easily the computing resources of hundreds of FPGAs. To demonstrate this last idea, we will validate the whole hardware-software framework presented in this chapter with an

application that shows how it can be used to develop a distributed cellular automaton and extended to less regular computations.

8.2 High-speed communication links

The first step required to be able to transmit information among the different CPUs present in the cell boards is to create local communication between the FPGAs of the routing layer, a step that must cope with a certain amount of constraints. First, the routing FPGAs possess only a limited number of wires (6) that can be used to implement high-speed communications in every direction, labeled North-East-South-West and Local (connected to the cell board) (see Figure 8.1).

Using only these links, the question is how to communicate efficiently and in full-duplex, i.e. so that both "sides" can communicate simultaneously, under the additional constraint that since scalability is a key feature of our architecture, no global clock is available to the system. This latter constraint implies that synchronization between the different FPGAs is not straightforward, notably to transmit information.

The first solution we tried was an *asynchronous* approach. However, because transmission delays can be relatively long in our system and only a few bits of parallel data can be used due to the limited number of wires, the use of a *request-acknowledge pair* in asynchronous transmission was too slow (early tests showed that the maximum bandwidth achievable using this technique was limited to about 50 $\mathrm{Mbit/s}$).

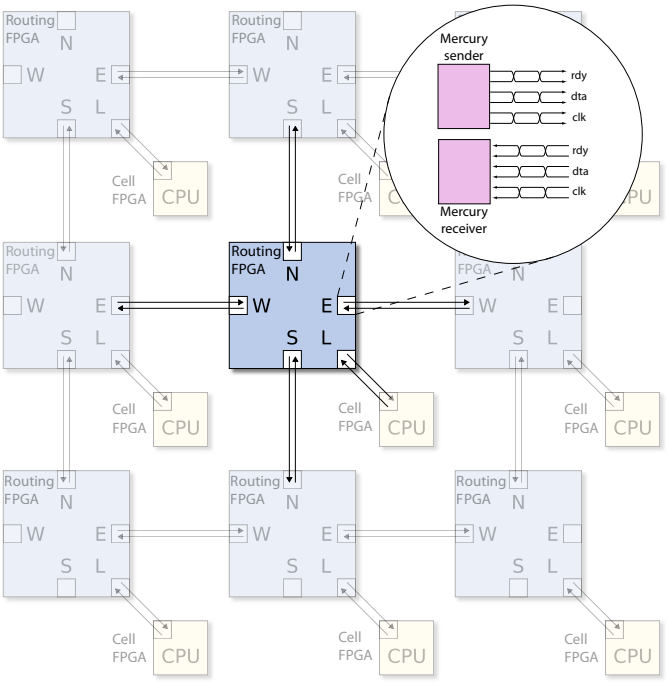
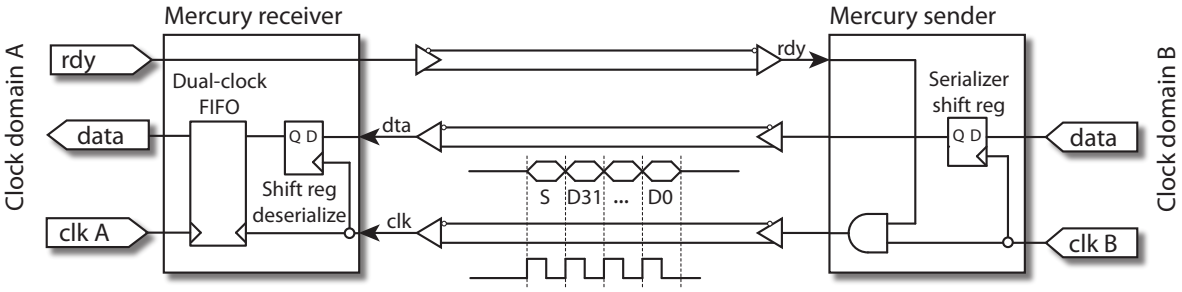


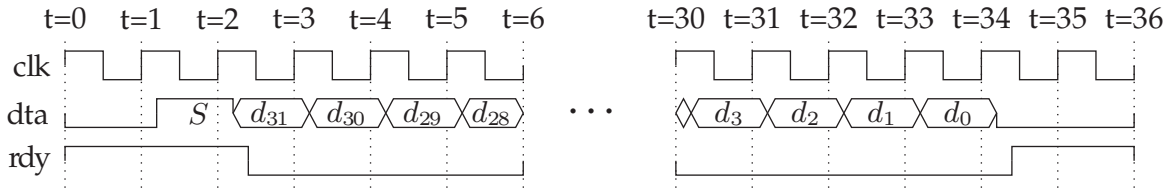
Figure 8.1: MERCURY high-speed links.

Depicted in Figure 8.1 and Figure 8.2, the solution retained is based on two modules: the first as a sender that transmits the clock along with serialized data to the second module that acts as a receiver. This receiver uses a dual-clock, dual-ported 32 bit FIFO queue to resynchronize the incoming flow of data to its own clock domain. With this approach, the transmission clock can be independent from the receiver's clock.

The steps of a transmission are as follows (Figure 8.2b): when it is ready for the data, the receiver asserts an acknowledge signal so that the sender can the transmit data synchronously with the clock signal it generates. The data begin with a start bit that indicates incoming data. If this solutions



(a) MERCURY sender and receiver modules, with external signals



(b) Timing

Figure 8.2: MERCURY sender and receiver hardware.

slightly reduces the usable bandwidth (1 bit out of 33 is wasted), it has the advantage to spare one physical wire that would have been otherwise required.

On the receiver side, once the data has been synchronized using the start bit, the incoming bits are deserialized in a shift register using the incoming clock signal. Once a complete word has been received, it is copied to the dual-clock FIFO which is used to cross the two different clock domains.

Electrically, LVDS lines are used to transport these high-speed signals, with the consequence that signal integrity is preserved over a longer distance at the cost of using twice the number of lines to transmit the signal.

In summary, this solution allows completely unrelated clocks, with different frequencies and/or phase shift, to be used in each clock domain and requires only three lines per link (using the FPGA differential I/O drivers) to implement full-duplex transmission in each direction.

A complete transceiver module, called MERCURY, is then composed of one receiver and one sender and can achieve a bandwidth of up to approximately 200 Mbit/s in full-duplex with a transmitting clock of 200 MHz. However, we prefer running it with a frequency of 100 MHz to limit the influence of electromagnetic interferences[1] (EMI). With this slower clock, the maximum data rate, accounting for overhead due to the start bit, is approximately 97 Mbit/s in each direction.

8.3 Routing data in hardware

Preliminary remark *Because the seminal work of Moraes et al. on the subject [Moraes 04] constitutes a first order source of information, this section is loosely based on that article, notably to introduce general information regarding packet-switching networks.*

While the above-mentioned high-speed links provide very local *physical* communication capabilities, the routing FPGAs obviously allow the implementation of more complex communication schemes such as broadcasting or point-to-point communication. According to different authors in

[1]Whereas the use of LVDS intent is precisely to attain a high tolerance for transmission, some impedance mismatch in the lines were discovered in the CONFETTI's routing PCB due to inadequate track lengths. For this reason, the system is relatively sensitive to high-frequency electrical noise such as the one generated by mobile phones in the 2.4 GHz spectrum. This translates to unreliable transmissions at high-speed in presence of mobile phones.

the domain [Benini 01, Benini 02, Dally 01, Kumar 02, Pande 05], the use of the *Network on Chip* (NoC) paradigm appears to emerge as a viable solution for the communication requirements of highly connected systems composed of tens or more of IP modules, notably to overcome the scalability and reusability issues of shared bus. The requirements for the connectivity of our platform are similar to the ones for NoC even if, technically speaking, the network in our architecture is not strictly *on-chip* can be extended *off-chip*.

Essentially consisting of connecting cores via a switched network, the NoC approach provides several advantages in terms of [Moraes 04]:

- energy efficiency and reliability;
- scalability of bandwidth with respect to standard bus architecture;
- reusability;
- distributed decisions for routing [Benini 02, Guerrier 00].

Thus, the intent of the NoC routing network we will develop in this section is to provide the necessary substrate to be able to implement, at the application level, complex data transfers between the cells via a simple network interface.

8.3.1 Basic features of NoC communication

In NoC, end-to-end communication is obtained with the exchange of messages between the different modules. Because messages can differ greatly based on the application, the notion of *packets* is preferred, because it allows to represent information that corresponds to a "fraction, one or even several messages" [Moraes 04, p. 3]. In most cases, packets are constituted of a header part that stores information regarding the rest of the message, called the payload, and sometimes a trailer part that terminates the message.

The network itself can be characterized through the *services* it provides and by the *communication system* it uses.

Services

The ISO OSI model [Day 83] is often used as a base to implement basic services such as data integrity or guarantees on network bandwidth or latency, which are enforced through quality-of-service (QoS) mechanisms. Examples of such mechanisms are fault-tolerance in communication [Dumitraş 03] and guarantees on such fault tolerance [Murali 06, Murali 07].

Communication system

This second NoC-defining parameter relates to *how* packets are transferred in the system, based on three main parameters:

1. **Network topology**
   Different layouts can be used to place the elements within the network, split into two categories: static and dynamic. In the first case, the switches have fixed connections and can be laid out in different topologies such as crossbar, n-dimensional k-ary mesh, fat-tree, torus, ... Dynamic layouts use buses or crossbars to change the connectivity of the network at runtime.

2. **Switching type**
   Circuit switching requires a preliminary phase to create a path before sending data whereas *packet switching* uses the packets themselves to create the connection. Several modes are available for packet switching depending on how the packets move between the switches. The most common are *store-and-forward*, *virtual cut-through* [Rijpkema 01] and *wormhole routing* [Ni 93]. These three techniques mostly differ in the way packets are buffered inside the switches.

3. **Routing algorithm**
   This determines how the paths taken by the packets are computed in the NoC. Starting from a source node, the path can be either determined completely at the beginning of the path (this is called *source routing*) or dynamically at each routing node. This last technique, called *distributed routing* [Duato 02], can then either always choose the same direction to route packets, depending on the source and destination addresses (*deterministic distributed routing*), or select the direction depending on the traffic (*adaptive routing*).

8.3.2 Brief existing NoC overview

The purpose of this brief overview is to give a quick outline of the differences in the various existing implementations of NoC to explain why the HERMES NoC was chosen for our system. For a much deeper discussion on the subject, we refer the interested user to [de Micheli 06].

As the NoC field itself is relatively young, some important contributions in the field focus on the very concept of NoC and on its importance [Benini 01, Dally 01, Ye 03], although they do not propose any implementations.

Most of the existing NoC systems found in the literature use packet switching (with the exception of aSOC [Liang 00]) with a clear predominance for mesh topologies, a fact that can be easily explained considering the simplified routing offered by this layout but also because this topology enables to scale a platform easily. A logical extension to the bi-dimensional mesh is the torus layout, in which the borders of the mesh are linked together. If this increases the wiring costs it also considerably reduces the network *diameter*[2] [Marescaux 02, Marescaux 03]. Another possibility to reduce the diameter – hence the latency – of the network is to use fat-tree or ring topologies, solutions chosen respectively by the SPIN network [Andriahantenaina 03] and the Octagon NoC [Karim 02].

If some NoC implementations clearly target ASIC (such as SPIN, aSOC [Liang 00] or Æthereal [Rijpkema 01]), FPGA NoC implementations that have been validated with a prototype are relatively scarce. Among others, Marescaux et al. [Marescaux 02, Marescaux 03] propose a NoC with a 2D torus topology that supports virtual channels. Bartic et al. [Bartic 05] have validated a NoC that supports arbitrary topologies through parameterizable links with congestion control. However, to our knowledge, only Moraes et al. offer accessible download of the source code of their HERMES NoC [Moraes 04].

8.3.3 The HERMES NoC

Of course, many different types of networking paradigms exist and could be implemented in our system, such as [Wiklund 03] or [Amde 05], two examples that cope with constraints such as real-time operation or asynchronous systems. However, when we read about a system closely resembling ours in [Ngouanga 06], we spent some time analyzing the HERMES NoC that was used. We found that this NoC provided simple and efficient routing along with a co-simulation environment that, thanks to the FLI library[3], permits C and VHDL communication in simulation. Moreover, the availability of the source code ensured that appropriate changes could be undertaken if necessary.

Due to all these advantages, the networking layer of CONFETTI uses a packet-switching algorithm loosely based on the HERMES framework, with the notable difference that the routing system implements the GALS paradigm [Teehan 07]: to avoid the issues inherent to large clock trees, only local (hence short) synchronous data transmission is used.

HERMES provides a network switch with five data ports. One is the *local* port, which interfaces the switch with the cell it belongs to. The others, the *external* ports, are connected to neighbor switches in the four cardinal directions. The role of the switch is to redirect packets, using the addresses embedded in the data headers, from any direction to any other direction, enabling point-to-point communication between *any* arbitrary pair of cell boards in the whole CONFETTI system.

[2] This is the longest distance from a point to another in the network.

[3] FLI stands for *Foreign Language Interface* and is a library developed by Mentor Graphics for its MODELSIM™ simulator.

Typically, a cell needs to send a message to another hands the data over to its associated switch through the local port. When receiving a message, a switch follows this policy:

- If the message has a destination address that does not match the switch's, it is routed towards its destination through an external port.
- If the message's destination address matches the switch's, it is sent through the local port and the cell receives it.

The very simple routing algorithm that the HERMES switches implement is known as *XY routing* (see Figure 8.3): nodes are given addresses of the form (x, y) which correspond to their physical coordinates on the grid. When routing, a message is sent towards its destination along the x axis, then once it is on the same x coordinate, along the y axis. The directions in which to forward the message can be computed by comparing x and y values between the switch's and the destination address.

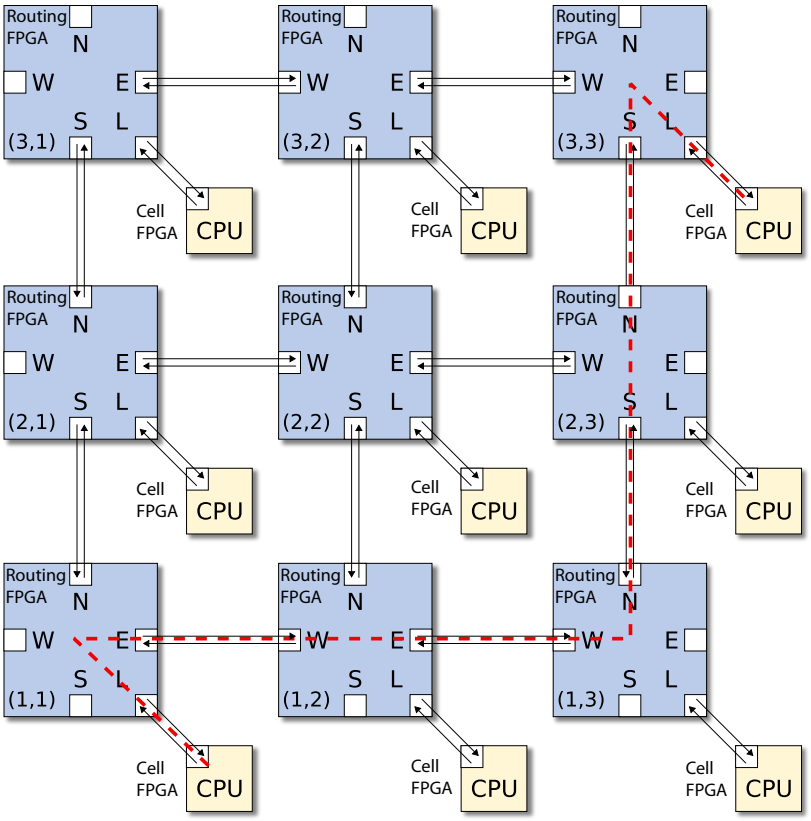


Figure 8.3: A 3x3 sample MERCURY network. The dotted line shows the path taken by a message from $(1, 1)$ to $(3, 3)$.

It should be noted that instead of trying to buffer entire messages before sending them (which would prove memory-consuming), HERMES uses *wormhole routing*. With this technique, messages are always transmitted in small units, or *flits*. The first flit of a message contains a header consisting of its destination d and length l: when a switch receives it, it routes the flit towards d through port p and stores (d, l). Port p is then reserved for the transmission until l flits come from the same port as the header and have been routed towards d, at which point the message has been completely transmitted.

On one hand, since it reserves an entire path for the duration of a message's transmission, it is more prone to congestion than packet-switched networks whose routers entirely buffer messages before retransmitting them. On the other hand, it requires much less buffer space and can be implemented directly in hardware.

8.3.4 MERCURY's additions to HERMES

One of the main reason HERMES was chosen over different routing algorithms was the availability of its source code so that some changes and additions could be applied to accommodate our needs. These changes are of two kinds:

1. **Serial protocol**
   HERMES allows some flexibility in the version we started with, regarding the number of wires used at low level to transport information. However, using a single line of data required to change the routing core to take into account serialization and deserialization of data.

   Moreover, HERMES initially used a FIFO to buffer a certain amount of flits during routing. However, the usage of a serial protocol, requiring multiple clock cycles to transmit a single word, allowed us to remove this internal FIFO to save hardware.

2. **Multiple clocks**
   The HERMES implementation relies heavily on the presence of single clock to synchronize its internal functioning. To accommodate our GALS platform, we had to insert adequate mechanisms to ensure correct behavior in presence of multiple clocks. This translated into inserting handshaking mechanisms in the routing elements of HERMES. Luckily, this was relatively straightforward as we could reuse some of the elements already present in HERMES and use them with the MERCURY transceivers that also possess handshaking signals.

Thus, in total, every constituting component of HERMES was modified: if the routing algorithm itself was not modified in its essence, every other file was changed to take into account the transceiver peculiarities as well as our clock requirements.

Routing performance analysis

Besides bandwidth measurements, no in-depth performance testing was undertaken with our implementation of HERMES, mostly because, qualitatively, the same performance figures obtained in [Moraes 04] are expected and also because the network performance has never been a limit to any of our applications, the processor being the real bottleneck of the system.

However, should a more detailed outline be required, methods used in [Coppola 04] or [Pande 05] could be applied.

8.4 Two additional functional units for ULYSSE

To take into account the functionalities of the CONFETTI platform and leverage its advantages in the ULYSSE processor, two functional units have been created: the first is a *high-speed communication unit* that interfaces the HERMES network through a MERCURY transceiver and the second is a unit to give ULYSSE access to the LED screen and to the touch sensitive area to enable user input and interactions.

8.4.1 The MERCURY FU

As mentioned earlier, every FPGA on the routing substrate is physically linked to its four cardinal neighbors and to the cell board above it (Figure 8.4) using MERCURY interfaces. Thanks to our revised HERMES implementation, every cell FPGA has now a connection to every other cell in the platform via a routing algorithm. The role of the FU described here is to interface the ULYSSE processor to that network.

For this purpose, a MERCURY transceiver is also used to communicate from ULYSSE to the routing FPGA (see Figure 8.4). Embedded inside a complete functional unit, the transceiver gives ULYSSE a direct access to the underlying routing algorithm for sending and receiving data. As this algorithm is clearly separated from the computation happening in the processor itself, testing and development of the module was relatively simple. Additionally, this clear separation also implies that different

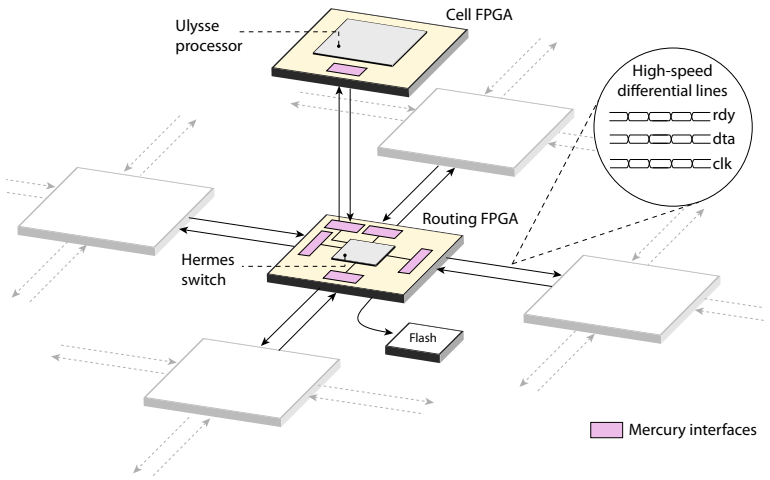


Figure 8.4: Detail of one routing FPGA and the link with its cell module.

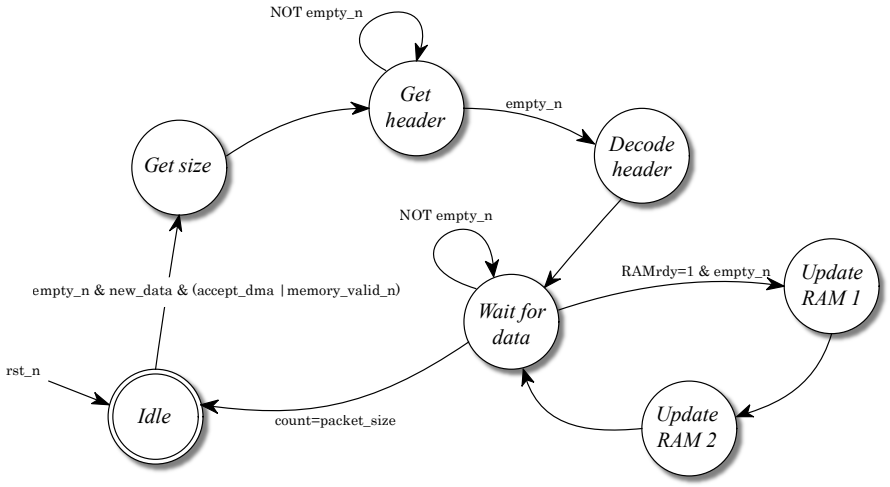


Figure 8.5: State machine of the receiver in ULYSSE MERCURY FU.

routing algorithms – be they based on HERMES or not – can be tested without changing this FU as long as the low-level interface relies on MERCURY.

Data sending from ULYSSE consists of moving the data to be sent to the register embedded in the FU. Reception is slightly more complex for performance reasons. In fact, we implemented a direct memory access (DMA) from the receiver to main memory so that incoming packets can be copied very fast from the interface to the memory. Without DMA, every word received would have required an explicit `move` operation from the FIFO to the memory, an operation that requires multiple displacements to setup and execute, which would have rendered reception unnecessarily slow. A packet reception is handled as follows (Figure 8.5): when new data arrives, if the unit is ready to accept new packets, the header of the data is first decoded to extract the number of flits composing the whole data stream; then, every time a new data word has been received and copied to memory (note that two cycles are required to perform this because memory access are 16-bit wide, see section 3.7.2, page 40), a counter is incremented to determine when the reception is complete. During the whole time the unit is waiting for new data to arrive, the processor is put in *stalled* state and the FU has direct access to the memory. Of course, this presents the disadvantage that incomplete packets block the processor, which could be a problem. This situation can be prevented by adding a timeout

counter to recover from such errors[4]. At the end of the reception, a flag register is updated to indicate that a new packet has been received.

Dynamic code reception

This unit also has a second function as it is also used to upload code to the processor. In fact, as the interface and the routing algorithm provide a convenient way to access *any* processor in the CONFETTI system from the outside using USB, it provides an interesting mean to upload processor code to be executed. For this reason, we added to the state machine adequate support[5] so that specially formed payload header trigger "CPU reprogramming". A message of this type, when received, is intercepted in hardware by MERCURY. Its payload is written in memory from address 0 onwards and replaces application code. This done, the CPU is restarted and the new code is executed.

Receiving code this way has the advantage that every component in the routing network – an ULYSSE processor or the PC connected via the USB interface, which also includes a MERCURY transceiver – is able to dynamically and quickly reprogram any processor in the computing layer.

Flow control

Additionally, control flow can be achieved thanks to the presence of the *status register* that permits to control how the unit reacts to incoming packets. For example, data reception can be inhibited by setting the adequate bit in the register. Thus, this register contains several control bits that have the following functions or meaning:

- `ACCEPT_DMA_MASK` (bit 0, lsb): 1 if the unit should accept new incoming data, if set to 0 the unit receiver keeps its `RDY` line deasserted;
- `ALLOW_RECONFIG` (bit 1): 1 if the unit is allowed to accept new code and reboot, if set to 0 the arrival of a packet with a new code will be ignored;
- `WRITE_IN_PROGRESS` (bit 2): 1 if MERCURY is currently receiving data;
- `NEW_DATA_FLAG` (bit 3): 1 if MERCURY receiver has finished writing a whole packet to memory;
- `CAN_SEND_DATA` (bit 4): 1 if MERCURY sender is ready to handle a new packet of data;
- `RECEIVER_EMPTY` (bit 5): 1 if there are data in the receiver waiting to be copied with DMA.

The complete memory map of the FU is summarized in Table 8.1.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	Status reg	The status register of the unit (7 bits used as described in 8.4.1).
1	RW	No	DMA Start	Start address of DMA for MERCURY to memory write.
2	RW	No	DMA End	Max address allowed for DMA write.
3	W	Yes	Rst	Any value written resets the new data flag.
3	R	No	Size	The size of the last received packet.
4	W	Yes	Send	Pass the written data to the MERCURY sender.
4	R	No	Header	Header of the last received packet.

Table 8.1: Mercury FU memory map.

[4]Because this never happened in our work, we did not implement this simple mechanism.
[5]This support consists in forcing the DMA start address to 0 and reset the CPU when the packet reception is complete.

8.4.2 Display and interface FU

As the routing board also connects to the LED display, ULYSSE CPUs also have access to the screen portion that lies above them through a dedicated functional unit which uses a slower serial protocol than MERCURY to communicate with the routing FPGA. Through this FU, each of the controllable 8x8 pixels can be accessed with its address and its color changed, using 8 bit values for each of the three basic color components (red-green-blue).

Additionally, through this FU the CPU can retrieve its X–Y coordinates in the global CONFETTI grid and sense the value of the touch-sensitive area glued on it. Other registers are also accessible, notably used to access the content of the Flash memory containing the FPGA bitstreams for the cell boards[6]. Thus, these bitstreams can be accessed and modified directly from ULYSSE, for example to be updated with new ULYSSE configuration or to implement a different processor than ULYSSE.

Table 8.2 reproduces the interface of this FU as seen by the processor. More details on its usage can be found in the annexed listing B.1, page 231.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	R	No	X	X-coordinate of this CPU in the CONFETTI grid.
1	R	No	Y	Y-coordinate of this CPU in the CONFETTI grid.
0	W	No	Addr	Address of the pixel to modify.
1	W	Yes	Color	Sets the color of the pixel.
2	R	No	Touch	# of seconds the touch sensor has been pressed continuously.
3	R	No	Graphic	Last pixel value written to the graphic memory.
[4...7]	RW	No	GR[0...3]	General purpose register [0...3].

Table 8.2: Display FU memory map.

8.5 A software API for routing

Preliminary remark *The forthcoming explanation on the software API is partly based, with his authorization, on Julien Ruffin's work [Ruffin 08] who did his master thesis on the subject under my supervision.*

Following the layered approach traditionally used in communication protocols, we implemented on top of the MERCURY physical layer a software *transport layer* that manages messages (Figure 8.6). The role of this component is to handle *packet* transmission and reception, providing more complex communication schemes than simple point-to-point communication. It consists of two essential parts: the first acts as a driver for MERCURY proper and provides simple send/receive functionality. Using it, the second part implements useful operations such as *broadcasting* or *guaranteed-delivery packets*. It also allows waiting for, then receiving, a packet fulfilling a set of given criteria.

8.5.1 Related work

As far as messaging layers are concerned, protocol stacks such as the one proposed by the *Nostrum* backbone [Millberg 04] offer functionalities that largely exceed our needs in this project. In comparison, the communication system shown in the *HS-Scale* project [Saint-Jean 07] provides interesting features that more closely resemble our needs, even though we designed simpler primitives than the data sockets it offers. In fact, HS-Scale makes heavy use of interrupts and their absence in the ULYSSE processor required this simplification. Notably, polling has to be used for I/O operations and static multitasking is obtained with simple cyclic executives, meaning that the processor always executes

[6]Low-level access to the Flash modules is handled by the routing FPGA and is not detailed here. We refer the interested reader to [Vannel 07].

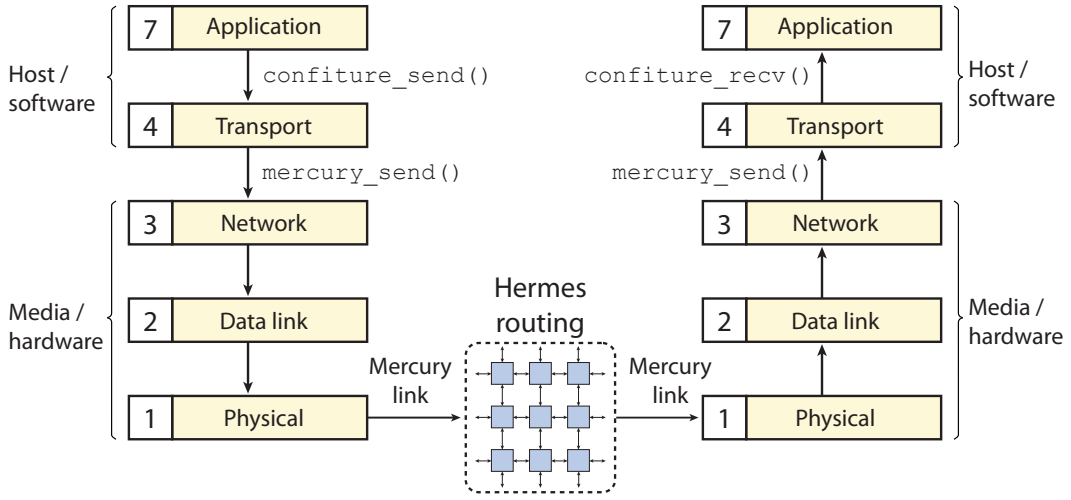


Figure 8.6: Communication organization, based on OSI layers.

a single computation task. As we will see, notably in the next chapter, this peculiarity will not be without consequences in terms of efficiency.

8.5.2 The CONFITURE communication infrastructure

To simplify the use of messaging, we created a base library that is used as a layer between the MERCURY low-level interface and user applications. Named CONFITURE, it implements a simple messaging system on top of the hardware and not only provides functionalities that derive from the hardware implementation but also adds its own special features.

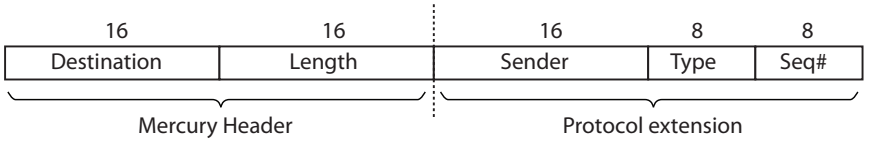


Figure 8.7: Message packet header format.

In practice, this layer extends MERCURY data packets – composed of a single word with destination and length of the packet – into its own format to include sender, message type and sequence number information (Figure 8.7). This header consists of the following fields:

- *destination* The destination address, formatted as $(x \ll 8) \parallel y$.
- *length* The length of the payload of the message, including the CONFITURE header extension.
- *sender* The address of the message sender, in the same format as the destination.
- *type* Message types that can have the following values, based on the behavior required:
  - *Asynchronous*, used for one-way messages. No reply is expected and waited by the library.
  - *Synchronous*, used for messages that need to be confirmed when received. When sent, the library function waits for an acknowledgment message from the recipient before considering the send successful.

  - *Broadcast*, used for messages that are propagated to the entire grid. These messages exist in two flavors, depending on their need to be acknowledged, i.e. a "synchronous" broadcast is considered correctly sent when all recipients acknowledge it.
  - *CPU reprogram*, a special message that when received is intercepted in hardware by the ULYSSE MERCURY FU. The payload of the message is then written to memory from address 0 onwards and replaces application code. This done, the CPU is restarted. Because of this, CONFITURE provides functions to generate such messages but not to receive them.

- *sequence number* A user-managed field that can be used for example to store sequence information to reorder packets. It can also be used to define different messages subtypes.

Functions

CONFITURE offers base `send()` and `receive()` functions in two different variants, blocking or non-blocking.

Another pair of functions are provided for cell reprogramming. One replicates the node's program to a given node address and the other, aptly named `confiture_spread()`, replicates the node's program over the *entire grid*. This latest feature provides a very efficient way to distribute a source code to the whole grid from the computer by reusing the same code and replicating it thanks to broadcast functions, which undeniably reduces reprogramming time because the whole routing substrate is used in parallel. The replication mechanism used will be described in detail later in section 10.4.2.

The last function provided by CONFITURE is `wait()`, which blocks until a message whose header fields match at least one *(sender, type, seq#)* tuple in a given list is received.

Even though we will limit our description of this library to this overview, the interested reader will find in the annex, section B.2, the complete API of the library.

It is worth noting that, feature-wise, the additions of CONFITURE are separated from the way the messages are handled at low-level. Consequently, the code is also separated into two sub-layers: one that contains the logic behind the additional features of CONFITURE[7] while the second acts a driver for MERCURY[8]. The advantage of this dual-layered approach is that CONFITURE could thus be ported to other networks than MERCURY by adapting the driver interface only.

Message queues

As mentioned earlier, an interesting constraint of the ULYSSE processor is its complete lack of interruption support. This, as well as the inability to reliably do context switching, prevents any easy use of multitasking. The main consequence of this is that running multiple applications or a micro-kernel based on timer interrupts is a non-trivial task on ULYSSE.

A common design used in messaging layers puts two message queues between the application and the messaging core proper. One of these queues is used for incoming messages and the other for outgoing messages; the application then reads in the first and writes in the second at will, the messaging system regularly filling the incoming queue and emptying the outgoing one, as long as there are messages present.

This works well in multitasking systems where the messaging layer can handle buffers transparently when needed, or in simpler systems using timer interrupts to periodically do the same. Since our system is neither multitasking nor able to use interrupts, any queue handling has to be manually called by the application. This imposes a restriction upon applications using the messaging layer: they must consist of a loop alternating their own computation and passing control to the layer.

Such a restriction is a serious drawback. Considering also the potentially large memory requirements of two queues, we decided to base CONFITURE on a radically different, simpler model. Getting rid of queues nearly completely, CONFITURE executes operations directly when called without message buffering, blocking until their completion.

[7] In file `confiture.h`, annex p. 242.
[8] In file `mercury.h`, annex p. 237.

All necessary handling – replying to "synchronous" messages, propagating broadcast messages and waiting for acknowledgments – is done as needed directly at either message send or reception, blocking until successful completion or error. When a broadcast is sent for the first time or a "synchronous" message is sent, the `send()` function waits for a corresponding reply. Whenever CONFITURE finds itself receiving a message from MERCURY that requires an acknowledgment in either the `receive()` and `wait()` functions, the needed replies are sent before the message is properly delivered.

As the layer implements `wait()` functions that block until the right message has been received, it is possible to receive messages while in that waiting state. Those messages are buffered in a small message queue dedicated to that purpose. The `receive()` function of CONFITURE first reads messages off this queue before querying MERCURY.

The broadcast algorithm

A naive way to implement broadcast would be to sequentially send the message to the whole node grid using a simple loop. Let N be the number of nodes in the grid: the same message then needs to be sent $(N - 1)$ times by the initiating node. With a growing N, this becomes not only hard to accomplish but extremely slow. The time needed to complete such a broadcast grows in $\mathcal{O}(N)$.

The solution we propose – and implemented – is more distributed. It uses two message types that we call *Level-1 broadcast* and *Level-2 broadcast*. To initiate a broadcast, a node sends the message to its immediate neighbors. On one axis of the grid, the node sends L1 broadcasts. On the axis orthogonal to it, L2 broadcasts are sent.

L2 broadcasts are propagated *along the same axis* by any node that receives it, as long as it stays in the limits of the grid. L1 broadcasts are propagated in the same way, with the addition that they also send L2 messages *along the orthogonal axis* in the likeness of the initiating node. Figure 8.8 illustrates the algorithm.

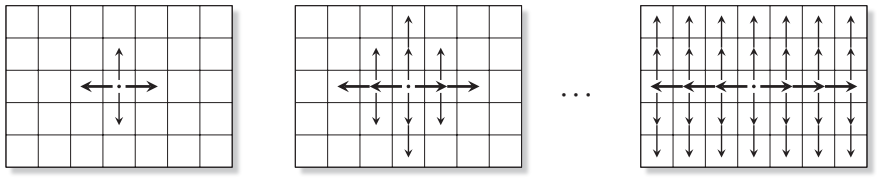


Figure 8.8: CONFITURE broadcast example. Thick and thin arrows represent (resp.) L1 and L2 broadcast messages.

This takes advantage of the mesh topology of CONFETTI's architecture. The average time needed for messages sent using this method to propagate themselves is by design directly related to the Manhattan distance between the origin of the broadcast and the farthest nodes on the grid, which grows in $\mathcal{O}\left(\sqrt{N}\right)$. We found square-root scalability sufficient for the desired applications and did not consider more complex solutions, although some, such as [Al-Dubai 02], exist.

Another possible solution we set aside is the propagation of broadcast messages along a growing square centered on the initial sender. While faster from a propagation time standpoint, it requires nodes at the corner of the square to propagate the message diagonally. This implies routing through another node in a direction to which a message was already sent, thus sending messages in the same direction twice, which is not desirable.

Note that our broadcast has a guaranteed-delivery ("synchronous") variant. With it, nodes that propagate the broadcast expect and wait for acknowledgment messages, sent back in the reverse path of the broadcast messages. As such, when the initiating node receives all acknowledgments, it is sure all messages have been successfully received by the entire grid. The "asynchronous" variant does not send acknowledgments and does not wait.

8.6 A sample application: simulating synchronicity with CAFCA

In this section, we validate our hardware–software design approach by showing how a traditional cellular automaton can be modeled within our framework. Cellular Automata (CA) date back to the beginnings of Computer Science [Von Neumann 66]. By construction, they are a form of parallel computing. The following formal description is quoted from the introduction of [Mitchell 98]:

> "A CA consists of two components. The first component is a *cellular space*: a lattice of N identical finite state machines (*cells*), each with an identical pattern of local connections to other cells for input and output, along with boundary conditions if the lattice is finite. Let Σ denote the set of states in a cell's finite state machine, and $k = |\Sigma|$ denote the number of states per cell. Each cell is denoted by an index i and its state at time t is denoted s_i^t (where $s_i^t \in \Sigma$). The state s_i^t of cell i together with the states of the cells to which cell i is connected is called the *neighborhood* η_i^t of cell i.
>
> The second component is a *transition rule* (or "CA rule") $\phi(\eta_i^t)$ that gives the update state $s_i^{(t+1)}$ for each cell i as a function of η_i^t.
>
> Typically in a CA a global clock provides an update signal for all cells: at each time step all cells update their states synchronously according to $\phi(\eta_i^t)$."

In other words, a CA is a set of identical cells aware of their state and that of their neighbors. At each time step, they use these data to update their state according to a transition rule. The variant we are going to consider uses a two-dimensional lattice of square cells and updates their states in a synchronous, deterministic manner.

The task is not trivial because of the lack of global synchronization clocks in the CONFETTI system. Typically, this absence would imply a considerable effort on the programmer's part to design and implement alternative synchronization mechanisms, again requiring a strong knowledge of VHDL and of the detailed operation of the hardware.

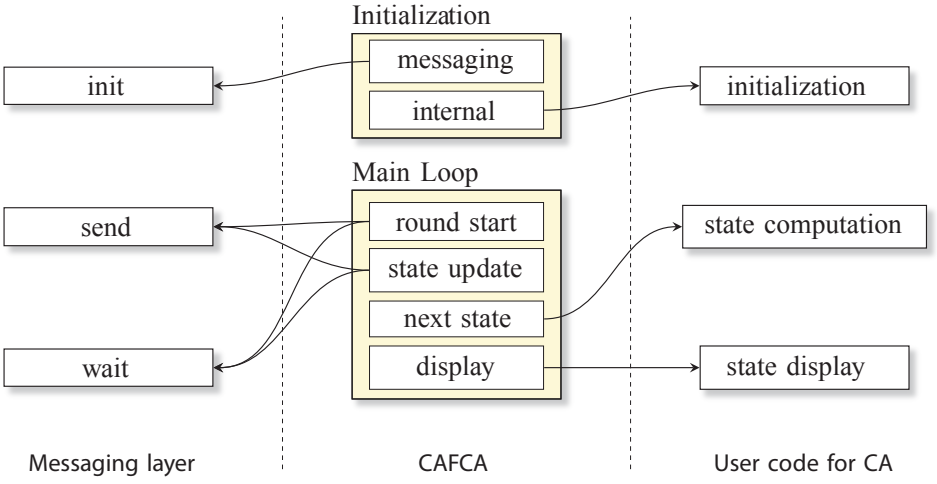


Figure 8.9: Structure of an application using CAFCA. Arrows represent function calls.

To overcome this limitation, we developed CAFCA (*Cellular Automata Framework for Cellular Architectures*), a software library that runs on the ULYSSE processor and that essentially permits, thanks to the messaging layer, to virtualize the fact that we are running a globally asynchronous system.

This tool represents a good example of the kind of software interfaces that can allow virtually any researcher to exploit the power of bio-inspired hardware implementations. In fact, when using this library it becomes then possible to develop parallel applications based on the cellular automata model by writing only a few C functions without any particular knowledge of the hardware. Because the development of the library had to be done only once, it greatly enhances the usability of the system.

To achieve a synchronized behavior, a step in the cellular automaton model implemented by CAFCA consists of four phases:

- Synchronization: all nodes wait for a broadcast message indicating the beginning of a computation round. If the node is the "coordinator" node (see below), it sends a synchronous broadcast to synchronize all the other cells.
- Inter-neighbor state exchange: information with the neighbors nodes is exchanged using simple asynchronous messages.
- Next state update: the value of the next state is computed from the node's own state and that of the neighbors.
- Display: the newly-computed state is sent to the LED display.

CAFCA builds upon the messaging layer and leverages the grid topology of the underlying hardware to provide cell synchronization and state exchange between neighbors that belong to different clock domains. Technically speaking, this is implemented with one of the cells, the "coordinator", playing the role of a supervisor to ensure, using broadcast messages, that every cell is at the same step.

Regarding the implementation, the coordinator's address and the initialization function pointer are the only CAFCA-specific arguments, respectively telling which node should be the coordinator and providing a pointer to the application's startup code. As a side note, it is possible to switch between Von Neumann and Moore neighborhoods with a compilation parameter. *Wrapping* is also provided by CAFCA. Thus, the cell space can be toroidal if needed: if a cell's neighbor happen to be outside the physical cell grid, the cell at the opposite end of the grid is accessed, "wrapping around" the coordinate axis. This feature is greatly facilitated by message routing.

An application that uses the framework library needs to provide only the functions for the computation and display phases, as well as emplacements to store local/neighbor states and their size, as shown on Figure 8.9. By fully handling synchronization and messaging issues, CAFCA enables fast development strictly centered on the two defining characteristics of a cellular automaton – the state variables and the state update function.

As an example, a simple implementation of Conway's Game of Life [Gardner 85] in our framework takes less than 100 lines of C code (corresponding to the user code in Figure 8.9). A complete version with 64 cells simulated per processor requires a total of about 200 lines of C code (the complete code is reproduced in the annex on page 245). A simplified version, showing a glider in 64 cells per processor, is shown in Listing 8.1.

Other examples we developed, such as a basic version of heat distribution in an homogeneous metal plate or the simulation of shallow water equations, can be coded in less than one thousand lines. Listing B.5 (p. 248) exemplifies the flexibility of the library by showing how different border conditions can be easily implemented. Figure 8.10 displays some pictures that were taken during the execution of various cellular automata[9].

8.7 Conclusion and future work

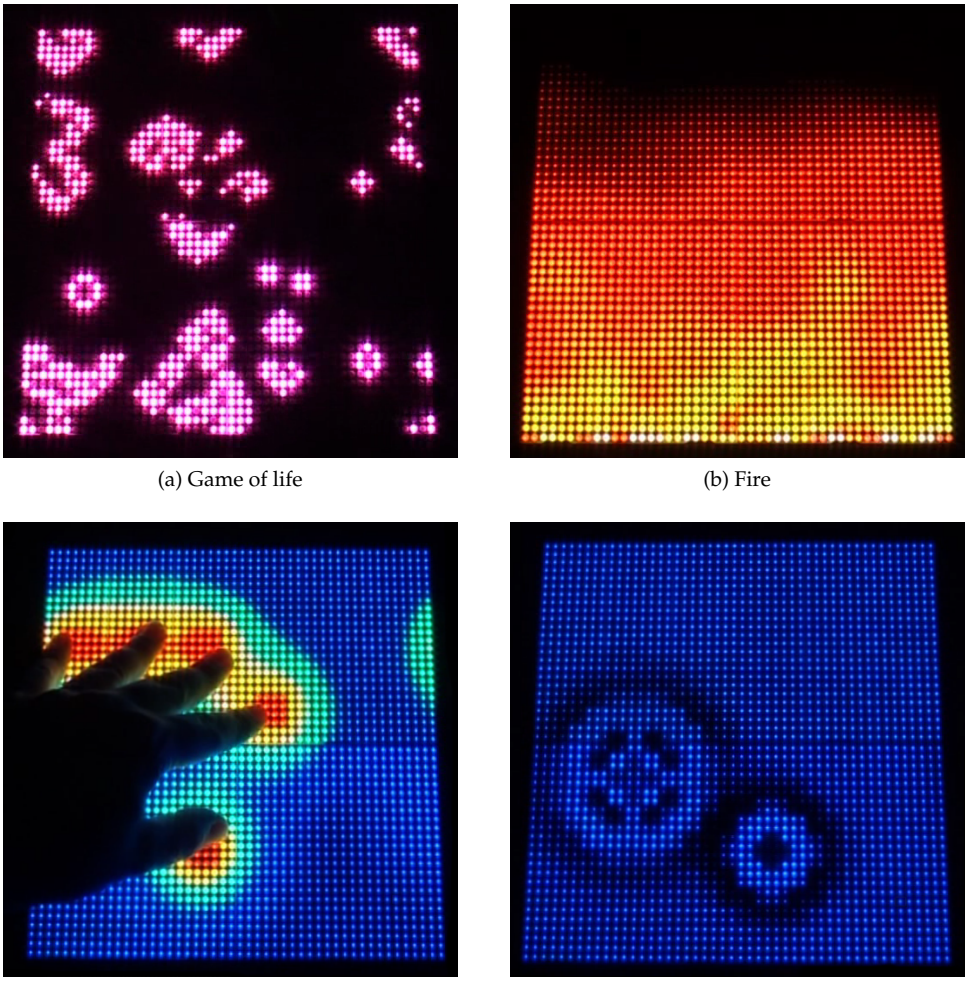
In addition to hardware routing in CONFETTI, which completes the description of our platform, we showed in this chapter a first type of application that can be realized with it.

As an example of the type of tools that could be invaluable for research in bio-inspired hardware, we showed how the framework we have developed can be used to provide an easy and rapid implementation of a traditional task for cellular hardware systems in the form of cellular automata. By developing several hardware and software components, we were able to drastically reduce the non-recurrent engineering time generally implied by the usage of intimately linked hardware and

[9] Videos showing some examples of the library usage on CONFETTI can be found at `http://birg.epfl.ch/page69673.html`

```
#include "cafca.h" /* compile with -DMOORE_NEIGHBORHOOD */

// A glider pattern
const int glider[8][8] = {
      {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}, {0,0,0,1,0,0,0,0}, {0,0,0,0,1,0,0,0},
      {0,0,1,1,1,0,0,0}, {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}};

void state_computation() {
    int weight, i, j;
    // Apply game-of-life rules
    for(i = 0 ; i < 8 ; i++) {
        for(j = 0 ; j < 8 ; j++) {
            weight = 0;
            weight += neighbour_wrapping(i,j+1); weight += neighbour_wrapping(i,j-1);
            weight += neighbour_wrapping(i+1,j+1); weight += neighbour_wrapping(i+1,j);
            weight += neighbour_wrapping(i+1,j-1); weight += neighbour_wrapping(i-1,j+1);
            weight += neighbour_wrapping(i-1,j); weight += neighbour_wrapping(i-1,j-1);

            if(state[i][j] == 0) {
                if(weight == 3) new_state[i][j] = 1;
                else new_state[i][j] = 0;
            } else {
                if(weight == 3 || weight == 2) new_state[i][j] = 1;
                else new_state[i][j] = 0;
            }
    }}}
    memcpy(state, new_state, 64); // Update the state
}

// Displays the state
void display_state() {
    int i,j;

    // Display the current state on screen, using purple color
    for(i = 0 ; i < 8 ; i++) {
        for(j = 0 ; j < 8 ; j++) {
            display(RED_PLANE + i*8 + j, state[i][j]*255);
            display(BLUE_PLANE + i*8 + j, state[i][j]*255);
    }}
}

void init() {
    int i,j;
    memset(state, 0, sizeof(state)); // Clear initial state
    // Copy the initial glider pattern on the grid
    if(local_addr == 0x0302)
        memcpy(state, glider, 64);

    app_state_size = 64; // Tell CAFCA the size of a state, in this case is a 8x8 grid
    app_state = (mercury_wd_t*) state; // Initialize the state at the beginning
    app_neighbor_st[0]=(mercury_wd_t*)n_state; app_neighbor_st[1]=(mercury_wd_t*)e_state;
    app_neighbor_st[2]=(mercury_wd_t*)s_state; app_neighbor_st[3]=(mercury_wd_t*)w_state;
    app_neighbor_st[4]=(mercury_wd_t*)ne_state; app_neighbor_st[5]=(mercury_wd_t*)se_state;
    app_neighbor_st[6]=(mercury_wd_t*)sw_state; app_neighbor_st[7]=(mercury_wd_t*)nw_state;

    // Sets the display and state functions used by the CAFCA API
    app_state_computation = &state_computation; app_state_display = &display_state;
}

void main() {
    // Initialize CAFCA with required parameters (size, etc.)
    cafca_init (0, 0x0101, &init, INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y);

    if(local_addr == 0x0202) confiture_spread(); // Distribute the code, starting from center
    cafca_main_loop(); // Launch the library main loop
    cafca_close(); // Tidy up when finished
}
```

Listing 8.1: Simplified code for Game-of-life implemented with CAFCA.



(a) Game of life

(b) Fire

(c) Heat dissipation

(d) Shallow water equation

Figure 8.10: CAFCA examples.

software. This was achieved by delegating all the complexity to a framework which presents to the programmer a simplified, yet powerful, view of the system. Thus, with a tool such as CAFCA, every programmer that knows C could exploit the computational power of hundreds of FPGAs with just a few lines of standard code.

A further development of the platform itself would be to implement *virtual channels* [Mello 05, Mello 06] to provide QoS guarantees on communication. Another interesting topic would consist of replacing ULYSSE with a more standard processor to analyze the impact of the TTA paradigm on performance at system level.

Regarding further developments of the software presented in this chapter, the implementation of other software libraries could, for example, take advantage of the reconfigurability of the system to implement growth or replication but also to realize computationally-intensive functions such as video decoding by implementing *ad-hoc* FUs.

In the next chapter, we will examine how different software tools can be designed to help easily develop applications for complex hardware platforms such as the one we just presented.

Bibliography

[Al-Dubai 02] A. Y. Al-Dubai, M. Ould-Khaoua & L. M. Mackenzie. *Towards a scalable broadcast in wormhole-switched mesh networks*. In Proceedings of the 2002 ACM symposium on Applied computing (SAC'02), pages 840–844, New York, USA, 2002. ACM.

[Amde 05] M. Amde, T. Felicijan, A. Efthymiou, D. Edwards & L. Lavagno. *Asynchronous On-Chip Networks*. IEE Proceedings Computers and Digital Techniques, vol. 152, no. 2, March 2005.

[Andriahantenaina 03] A. Andriahantenaina & A. Greiner. *Micro-network for SoC: implementation of a 32-port SPIN network*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), pages 1128–1129, 2003.

[Bartic 05] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde & R. Lauwereins. *Topology adaptive network-on-chip design and implementation*. IEE Proceedings – Computers and Digital Techniques, vol. 152, no. 4, pages 467–472, 2005.

[Benini 01] Luca Benini & Giovanni de Micheli. *Power networks on chips: energy-efficient and reliable interconnect design for SoCs*. In Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01), pages 33–38, October 2001.

[Benini 02] Luca Benini & Giovanni de Micheli. *Networks on Chips: A New SoC Paradigm*. Computer, vol. 35, no. 1, pages 70–78, 2002.

[Coppola 04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia & Francesco Papariello. *OCCN: a NoC modeling framework for design exploration*. Journal of Systems Architecture, vol. 50, no. 2-3, pages 129–163, 2004.

[Dally 01] William J. Dally & Brian Towles. *Route packets, not wires: on-chip inteconnection networks*. In DAC '01: Proc. 38th Conf. on Design automation, pages 684–689, New York, USA, 2001. ACM Press.

[Day 83] John D. Day & Hubert Zimmermann. *The OSI reference model*. Proceedings of the IEEE, vol. 71, no. 12, pages 1334–1340, 1983.

[de Micheli 06] Giovanni de Micheli & Luca Benini. *Networks on Chips: Technology and Tools (Systems on Silicon)*. Morgan Kaufmann, first edition, 2006.

[Duato 02] Jose Duato, Sudhakar Yalamanchili & Lionel Ni. *Interconnection Networks: An Engineering Approach*. Elsevier Science, revised edition, 2002.

[Dumitraş 03] Tudor Dumitraş, Sam Kerner & Radu Mărculescu. *Towards on-chip fault-tolerant communication*. In ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation, pages 225–232, New York, USA, 2003. ACM.

[Gardner 85] Martin Gardner. *Wheels, Life, and Other Mathematical Amusements*. W.H. Freeman & Company, 1985.

[Guerrier 00] P. Guerrier & A. Greiner. *A generic architecture for on-chip packet-switched interconnections*. In Proceedings of the conference on Design, automation and test in Europe (DATE'00), pages 250–256, 2000.

[Karim 02] F. Karim, A. Nguyen & S. Dey. *An interconnect architecture for networking systems on chips*. IEEE Micro, vol. 22, no. 5, pages 36–45, 2002.

[Kumar 02] S. Kumar, A. Jantsch, J.P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja & A. Hemani. *A network on chip architecture and design methodology*. In IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), pages 105–112, April 2002.

[Liang 00] Jian Liang, Sriram Swaminathan & Russell Tessier. *aSOC: A Scalable, Single-Chip Communications Architecture*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, vol. 0, pages 37–46, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[Marescaux 02] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde & Rudy Lauwereins. *Interconnection Networks Enable Fine-Grain Dynamic Multitasking on FPGAs*. In Proceedings of 12th International Conference on Field-Programmable Logic and Applications (FPL '02) - The Reconfigurable Computing Is Going Mainstream, pages 795–805, London, UK, 2002. Springer-Verlag.

[Marescaux 03] Théodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, W. Moffat, Diederik Verkest, Serge Vernalde & Rudy Lauwereins. *Networks on Chip as Hardware Components of an OS for Reconfigurable Systems*. In Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL'03), pages 595–605, 2003.

[Mello 05] Aline Mello, Leonel Tedesco, Ney Calazans & Fernando Moraes. *Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC*. In Proceedings of the 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), pages 178–183. ACM, 2005.

[Mello 06] Aline Mello, Leonel Tedesco, Ney Calazans & Fernando Moraes. *Evaluation of current QoS Mechanisms in Networks on Chip*. In Proceedings of the International Symposium on System-on-Chip, pages 1–4, November 2006.

[Millberg 04] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar & Axel Jantsch. *The Nostrum Backbone - a Communication Protocol Stack for Networks on Chip*. In Proceedings of the 17th International Conference on VLSI Design (VLSID'04), page 693, Washington, USA, 2004. IEEE Computer Society.

[Mitchell 98] Melanie Mitchell. *Computation in Cellular Automata: A Selected Review*. In Nonstandard Computation, pages 95–140. VCH Verlagsgesellschaft, 1998.

[Moraes 04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller & Luciano Ost. *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. Integrated VLSI Journal, vol. 38, no. 1, pages 69–93, 2004.

[Murali 06] Srinivasan Murali, David Atienza, Luca Benini & Giovanni de Micheli. *A Multi-Path Routing Strategy with Guaranteed In-order Packet Delivery and Fault Tolerance for Networks on Chips*. In Proceedings of the Design Automation Conference (DAC'06), pages 845–848, 2006.

[Murali 07] Srinivasan Murali, David Atienza, Luca Benini & Giovanni de Micheli. *A method for routing packets across multiple paths in NoCs with in-order delivery and fault-tolerance guarantees*. VLSI Design, vol. 2007, page 11, 2007.

[Ngouanga 06] Alex Ngouanga, Gilles Sassatelli, Lionel Torres, Thierry Gil, André Soares & Altamiro Susin. *A contextual resources use: a proof of concept through the APACHES' platform*. In Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pages 44–49, April 2006.

[Ni 93] Lionel M. Ni & Philip K. McKinley. *A Survey of Wormhole Routing Techniques in Direct Networks*. IEEE Computer, vol. 26, no. 2, pages 62–76, 1993.

[Pande 05] Partha Pratim Pande, Cristian Grecu, André Ivanov, Resve Saleh & Giovanni de Micheli. *Design, Synthesis, and Test of Network on Chips*. IEEE Design & Test of Computers, vol. 22, no. 5, pages 404–413, 2005.

[Rijpkema 01] Edwin Rijpkema, Kees Goossens & Paul Wielage. *A Router Architecture for Networks on Silicon*. In Proceedings of the 2nd Workshop on Embedded Systems (Progress'01), pages 181–188, 2001.

[Ruffin 08] Julien Ruffin. *Self-organized Parallel Computation on the CONFETTI Cellular Architecture*. Master's thesis, École Polytechnique Fédérale de Lausanne, 2008.

[Saint-Jean 07] Nicolas Saint-Jean, Gilles Sassatelli, Pascal Benoit, Lionel Torres & Michel Robert. *HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems*. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07), pages 21–28, 2007.

[Teehan 07] Paul Teehan, Mark Greenstreet & Guy Lemieux. *A Survey and Taxonomy of GALS Design Styles*. IEEE Design and Test, vol. 24, no. 5, pages 418–428, 2007.

[Vannel 07] Fabien Vannel. *Bio-inspired cellular machines: towards a new electronic paper architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, 2007.

[Von Neumann 66] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[Wiklund 03] Daniel Wiklund & Dake Liu. *SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems*. In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03), page 78.1, Washington, USA, 2003. IEEE Computer Society.

[Ye 03] Terry Tao Ye, Luca Benini & Giovanni de Micheli. *Packetized On-Chip Interconnect Communication Analysis for MPSoC*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), vol. 1, pages 344–349, Los Alamitos, CA, USA, March 2003. IEEE Computer Society.

Chapter 9

A software environment for cellular computing

"A worker may be the hammer's master, but the hammer still prevails.
A tool knows exactly how it is meant to be handled,
while the user of the tool can only have an approximate idea."

MILAN KUNDERA, *The Book of Laughter and Forgetting*

THE APPLICATION of bio-inspired mechanisms such as evolution, growth or self-repair in hardware requires resources (fault-detection logic, self-replication mechanisms, ...) that are normally not available in off-the-shelf circuits. For these reasons, a number of dedicated hardware devices were developed, e.g. [Greensted 07, Tempesti 01, Thoma 04, Upegui 07], with the intent of exploring various bio-inspired paradigms.

In general however, they represent experimental platforms that are very difficult to program and require an in-depth understanding of the underlying hardware. As a consequence, these platforms are accessible only to a limited class of programmers who are well-versed in hardware description languages (such as VHDL) and who are willing to invest considerable time in learning how to design hardware for a specific, often ill-documented, device.

Notwithstanding these issues, hardware remains an interesting option in this research domain as it can greatly accelerate some operations and because it allows a direct interaction with the environment. It is however undeniable that the difficulty of efficiently programming hardware platforms has prevented their use for complex real-world applications. In turn, the fact that experiments have been mostly limited to simple demonstrators has hindered the widespread acceptance of the bio-inspired techniques they were meant to illustrate.

While, by their very nature, bio-inspired systems rely on non-conventional mechanisms, in the majority of cases they bear some degree of similarity to networks of computational nodes, a structure that is frequently used when dealing with parallel computing systems. In these latter, a software abstraction is often used to hide the complexity and details of the underlying hardware and simplify programming. It is therefore legitimate to wonder if such an approach can be applied to bio-inspired hardware systems as well, to allow researchers to rapidly prototype new ideas and, more importantly, to cope with the complexity of tens or hundreds of parallel computational elements.

In this chapter we describe a set of software tools, gathered in a graphical user interface (GUI), that attempt to help the user with intuitive hints during the critical phases of the design of distributed cellular architectures. More precisely, we will present a software framework that provides a model of systems such as CONFETTI to simplify their programming thanks to a strict separation of the hardware and software. We will see that with this framework, deliberately open and modular so that it can be easily extended with new modules, it becomes possible to quickly evaluate performance, algorithms, techniques and ideas for the community of researchers interested in bio-inspired hardware.

9.1 Introduction and motivations

Although the hardware in almost every desktop computer sold nowadays offers the possibility of executing multiple threads in parallel, a majority of the software written for these machines remains serial. This situation can be explained, at least partly, by the complexity of writing multithreaded applications: when multiple threads run in concert, the sequential model of programming breaks down. For example, the apparition of shared variables necessarily translates to concurrent accesses, which in turn requires to analyze program functions with a broader view than usual, the programmer having to think outside the scope of the function.

Of course, several programming languages propose techniques to reduce the complexity penalty of introducing parallelism: for instance, functional languages such as *Haskell*[1] [Hutton 07] do not include the notion of shared state nor that one of sequential execution, and other languages, such as APL [Iverson 62], remove the limitation of doing a single thing at a time. However, all these languages are not widespread and they lack the acceptance and backing of the C/C++ or *Java* languages. Thus, the solution generally used to help parallel programming with more "traditional" languages relies on libraries, such as the *Message Passing Interface* (MPI) [Snir 96], that provide an *abstraction layer* between the parallel system where the application is executed and its software counterpart.

One of the stated objectives of this thesis is to generalize the use of bio-inspired techniques, notably in the context of cellular computing. As we have already mentioned, this latter bears some similarities with parallel programming, although with notable differences in terms of resource available when compared to desktop computing but also in terms of OS stability, desktop computers benefiting from software with years of development behind them. Thus, the direct translation of such a library is difficult or even impossible, notably due to the heavy resource constraints present in embedded systems.

In our opinion, the development of an approach similar to the one proposed by *MPI* could be helpful for cellular computing in embedded systems: in fact, the need of an intermediate layer to create distance between the low-level hardware and the software representation of that hardware is even more acute in such systems because they often lack a proper operating system that plays this role.

In this chapter, we present such a software for our cellular approach, with the main objective of proposing an *unified environment* for cellular computing so that any user can rapidly handle complex parallel computing platforms without having to master the nitty-gritty details of that hardware. The secondary objective of the interface-based program is to maintain a high level of flexibility so that new components can be easily added to accommodate the needs of different users, different applications and rapidly prototype new ideas. In other words, our intention in this chapter is to propose tools to help embedded developers tackle the problems that arise concurrently with the advent of parallel embedded systems. To achieve this goal we will proceed as follows: in the next section we will present and justify the validity of the program model we use, which consists in specifying applications as a graph of tasks using a visual interface. We will then present a general overview of the application in section 9.4, which articulates around two distinct sets of tools. The first one, aimed at helping the user create the core functions of the programs, will be detailed in section 9.5. The second set of tools provides support for developing and analyzing task graphs, notably thanks to two different graph simulation environments.

9.2 Graphical programming languages

Visual or graphical programming languages let users create applications by *manipulating* visually the constituting elements of a program where a traditional specification would require a textual approach. Thus, the majority of these languages use graphical objects connected together to represent the relations between the various program entities. Often used to describe systems with concurrency issues, visual languages such as *Petri nets* [Peterson 81] have been notably used to model parallel systems,

[1] See `http://www.haskell.org`

tackling issues such as resource sharing. More recently, these languages have shown a revamped interest because they can be used to represent efficiently dataflow programming (see section 9.3.2), a paradigm which has been identified as a promising candidate for the automatic parallelization of programs [Johnston 04]. In the context of this thesis, the advantage of using such a visual programming language is three-fold:

1. *simplify the development* of parallel programs by providing a visual and intuitive way to express applications in terms of objects linked together;

2. provide a *visual feedback* of the performance and behavior of the application;

3. *create a clear distinction* between the software, which is expressed visually to illustrate abstractions, and the hardware itself, which is absent from that visual description.

Naturally, the idea of using a GUI interface for studying and creating parallel programs is not, in itself, new. For instance, in 1990, Lewis et al. [Lewis 90] proposed a tool for scheduling large-grain parallel program tasks under the assumption that *"the best solutions are obtained in cooperation with a human designer"* [Lewis 90, p. 1172]. This statement is based on the observation that, due to the very different nature of existing problems that might have to be solved by parallel systems, relying to some extent on human expertise during the design cycle is a good idea, an opinion that we share. As a result, the software GUI we propose contains several components that help the user make good decisions, for example with tools to graphically model parallel programs using task graphs or with a tool to observe different performance estimates of the program.

Regarding the need for a distinct separation between hardware and software, recent works underline the importance of the concept, notably for embedded systems. Thus, Engler et al. [Engler 95] or Nesbit et al. [Nesbit 08] have shown the advantages of using the notion of *virtual machines* with which a software layer can be used to implement policies whilst the hardware provides mechanisms to support them. Even though we will not use the virtual machine paradigm in this work, the approach we use bears some similarities with the concept in that it uses an interface between hardware and software, an approach also proposed for example in the work of van der Wolf et al. [van der Wolf 04], who stress the importance of such a layer:

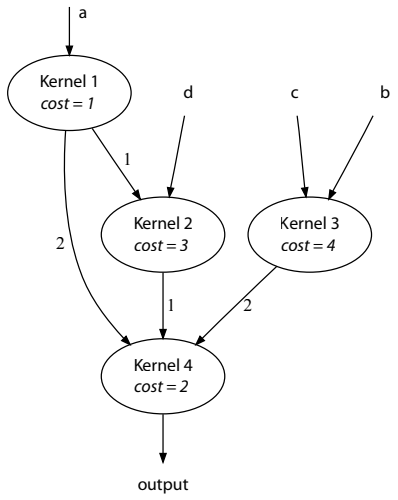
> "[this interface] should help to close the gap between the application models used for specification and the optimized implementation of the application on a multi-processor architecture [...] Such interface and mapping technology will help to structure MPSoC integration, thereby enhancing both the productivity and the quality of MPSoC design." [van der Wolf 04, p. 206]

9.3 A programming model

If visual programming languages allow to manipulate programs, a *model* for these programs is still required. Luckily, the literature is rich with different solutions for this question. The one we use in this work is one of the most commonly used, i.e. the *directed acyclic graph* (DAG) model. Even though other interesting alternatives can also be used, such as Kahn process networks [Kahn 74], DAGs are simple enough to model our needs and are also more widespread, which implies that the considerable work done on this kind of applications could also be applied to suit our current or future needs.

9.3.1 Directed acyclic graphs programs

Definition A *directed acyclic graph program* is a directed graph without cycles in which vertices represent tasks, characterized by an execution cost, whose connections to other vertices through edges represent precedence relations, potentially characterized by a communication cost [Sarkar 89].



(a) DAG

```
// Kernel_1
void compute(input, output){
  output[0] = input[0] + 10;
}

// Kernel_2
void compute(input, output){
  output[0] = input[0] * input[1];
}

// Kernel_3
void compute(input, output){
  output[0] = pow(input[0], input(1));
}

// Kernel_4
void compute(input, output){
  output[0] = input[0] + input[1] - input[2];

}
```

(b) Corresponding code

Figure 9.1: Sample of an application in DAG form with its corresponding pseudo-code.

Figure 9.1 shows how this graph representation can be used to model a simple parallel program that computes the value of $a+(a+10)\cdot d-(b^c)$. In the graphic representation, each node[2] is represented as an ellipse and the execution time of the task is written inside it, whereas the communication cost is written on each edge. In this model, a task is triggered (i.e. starts working) when all its predecessors have produced an output that is then used as an input for the successor node (also known as the *asynchrony* property, see [Gajski 82]). The second property of the model is called the *functionality* property (see [Gajski 82]): as every operation is a function and no shared memory is used, side effects are impossible, i.e. no state is modified in addition to returning a value.

DAGs have been used in the literature to examine notions such as *scheduling*, a NP-hard problem [Garey 79] which consists in minimizing the total execution time (also called *makespan* or *schedule length*) of parallel programs, by allocating tasks to processors so that the precedence order is maintained [Ahmad 99]. However, if DAGs can be used to describe the relations between the tasks, they do not stipulate how the application itself is programmed. As such, they can be considered as a *conceptual modeling* tool for parallel programs to which a *programming model* should be added to be used in real situations and that will be examined in the next subsection.

9.3.2 Dataflow / stream programming

The dataflow programming paradigm is a form of parallel processing that simplifies the development of certain types of applications, at the expense of some limitations. This model makes use of *kernel functions*, represented with a DAG, that are applied to a set of data called the *data stream*. In this DAG, the vertices represent the kernel functions and the edges account for the precedence relations taken by the data stream.

The *stream programming* paradigm closely resembles the dataflow paradigm, with the notable difference that the former tends to consider computation at a lower lever, closer to the machine. In the remainder of this work, these two concepts should be seen as synonyms, even though a difference exists in terms of the implementation of the model.

Multiple languages were developed with these paradigms as a cornerstone, such as the *StreamIT* language at MIT [Thies 02, Gordon 06], the *Brook* language at Stanford [Buck 04] or Intel's *Ct* [Ghuloum 07] (which is a proprietary language). These languages were mainly developed for specific so-called *stream* processors, such as the Stanford *Imagine* [Kapasi 02] and *Merrimac* [Dally 03], the MIT *RAW* [Taylor 02] or to a certain extent the Sony-Toshiba-IBM *Cell* [Pham 06]. However, the scope of

[2] In this thesis, the terms *tasks* and *nodes* have the same meaning and are used indifferently.

these languages can be extended to model a whole range of execution platforms, such as the graphic processing units (GPU) used in graphic cards, which also use a similar programming model in their shader language (for instance, *Cg* [Fernando 03]). When it comes to the specific domain of MPSoC, the work of Jerraya et al. is of interest [Jerraya 05, Jerraya 06].

The wide and increasing acceptance of the model comes from a flexibility that enables it to be used in a whole range of situations. Another advantage in its favor is that the kernel functions work *locally* on *independent* data. These two facts greatly simplify the development of parallel programs because they allow to bring back a more serial model, which helps maintaining a good overview of the problem tackled by the application. Moreover, another advantage of stream programming resides in the fact that the definition of the problem is relatively well separated from its resolution. Thus, application writing becomes a two-fold process:

- During the first step, when the task graph is defined, the programmer can focus on splitting the application in different, parallel tasks. When doing so, an abstract view of the data transmitted between the tasks can be kept. Because this separation helps keeping a *linear and sequential view* of the application, the development is simpler, closely resembling non-parallel code programming.
- The second step consists in programming the tasks present in the graph. Thanks to the explicit parallelism of the graph, each task separated from the others receives a clear, distinct input and has to produce an output with similar characteristics. The function of the task is thus to produce, starting from an input data stream, an output data stream that has been generated according to the role of the task. The advantage of this clear role of the task is that the programming becomes completely sequential: parallelism during the programming of the task can be abstract or at least limited to lower-level parallelism, such as instruction-level parallelism.

Finally, another advantage of the task model is that synchronization, allocation and communication do not need to be explicitly defined in the application code. These details can be hidden from the end-user, a property which can be quite useful, as we will see in the remainder of this chapter.

Of course, all these advantages come at a cost and some limitations are inherent to the model, which cannot be applied to every kind of parallel program:

1. The communication cost as well as the execution time are not constant in every program and may greatly vary during execution. If this information is used, for example by scheduling algorithms, the user must ensure to model it correctly.
2. Not every application can be described as a task graph, especially without cycles. For instance, the development of digital filters with loops is not directly possible.
3. The approach requires determining a certain *granularity* for parallelism to achieve the appropriate balance between computation and communication. This determination of the appropriate parallelism level – task, function, data – is a complex question that can greatly influence the quality of the final solution in terms of performance.

Despite these limitations, the model is sufficient for our requirements and provides enough advantages to reduce the impact of these drawbacks, as we will see.

9.4 Software framework overview

Through a convenient graphical user interface, the programming environment we designed is dedicated to helping the user in every step of the development of a cellular program. The GUI itself is written in *Java* and uses the *Prefuse* library[3] [Heer 05] to represent and layout graphs.

[3]This free visualization library, developed at Berkeley, is tailored at creating dynamic visualizations for both structured and unstructured data. More info on `http://www.prefuse.org`

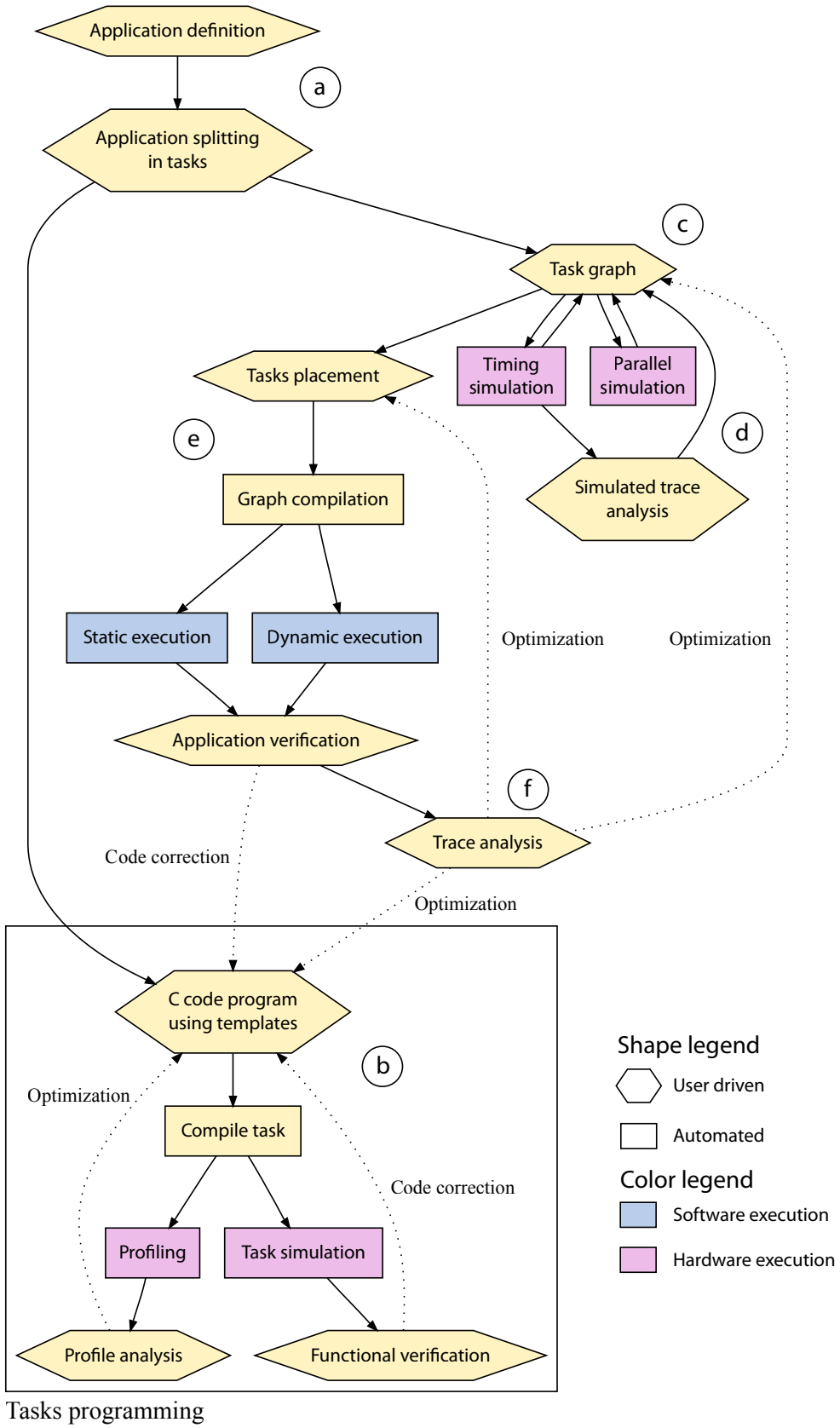


Figure 9.2: Development environment for cellular programming.

A general overview of the functionalities of the applications is depicted in Figure 9.2, which shows how the different phases of the programming process interact[4]. Following that diagram, the outline of a typical programming session in the GUI is as follows:

1. For each application, the user starts by manually analyzing and splitting the application into tasks (a), using the methodology presented in section 9.3.1.
2. Each task defined in the application splitting phase is programmed using the C language (b). The tasks can also be *profiled and simulated* to gather performance estimates.
3. Using the different tasks, the application graph can then be created using a visual programming approach (c).
4. The graph generated can then be simulated (d), placed and downloaded (e) to the hardware platform.
5. Using the hardware platform, results and performance measures can be gathered on the computer and displayed (f). Optionally, the user can decide to use the dynamic replication approach that will be introduced in the next chapter.

The framework helps the user during most of these phases with different tools, that can be divided into two major categories. The first set relates to tools that help the development of the tasks themselves whereas the second consists of tools that help the development and the execution of the graphs. Both sets will be examined in details, starting with the task tools.

9.5 Task programming tools

Represented in the bottom left-hand corner of Figure 9.2, the task development tools can be used for programming, simulating, profiling and debugging the kernel functions. Let us now consider each of these functions, starting with the programming.

9.5.1 Task programming using templates

Writing the tasks' code uses a set of *C code templates* proposed within the GUI environment. These templates, which consist of application skeletons composed of different functions that the user has to fill, are intimately related to the kind of computation to be undertaken. Thus, different templates exist for different types of applications.

For instance, a code template for the CAFCA library (presented in section 8.6, page 132), is provided in the annex page 254. With this template, the user has to fill the functions required for displaying the state of the cellular automata and for computing the next state. This "code filling" of the task can be done using the integrated text editor, shown in Figure 9.3, that provides commonly-found features in such editors, such as syntax highlighting or code folding.

At the time of writing, two distinct code templates have been developed: the one just presented which is tailored for cellular automata and a second for the SSSP dynamic replication environment, which will be examined in the next chapter (chapter 10). Generally speaking, these templates consist of pre-defined functions which have the following targets:

1. *Increase the productivity* of the user, simplify the programming and also limit the risks of errors. In fact, with templates, since the user "only" has to fill the recurrent functions that must be present in every task, the programming process is simplified so that the user can focus on the aim of each kernel without having to cope with the general behavior of the application. With templates, it becomes possible to "hide" some of the complexity to the end-user, which constitutes an

[4]We also refer the interested reader to the page 253 of the appendix, where the complete architecture of GUI software is presented.

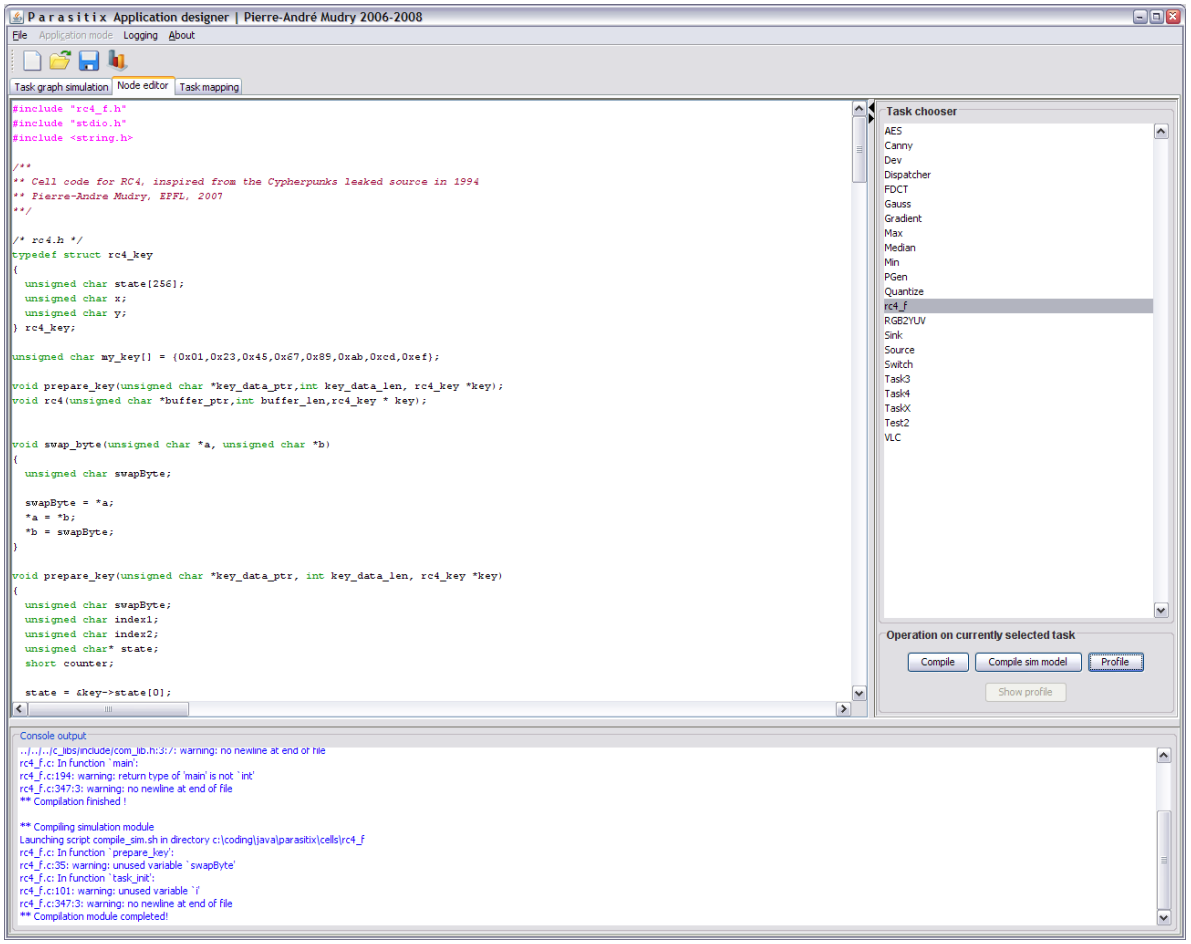


Figure 9.3: The GUI text code editor.

advantage because more time can be devoted to the core application itself. For example, when using the CAFCA template, the user does not need to handle the complexity of synchronizing all the processors nor to care about sending and receiving states from the neighbors. In this case, by conforming to a template, the user can rely on a library to do it.

2. As the tasks code is used by the tools in different contexts (simulation, profiling, ...), templates are convenient to ensure that every developed task presents a similar interface that can be *manipulated in software*. For example, templates developed thus far provide different functions for profiling and execution in simulation, which makes possible to automatically generate different performance graphs (see section 9.5.4) but also to debug and simulate the code (see section 9.5.3).

3. New templates can be easily developed, as the GUI supports the modification and the addition of templates. Moreover, the templates themselves provide some flexibility. Thus, some keywords can be used in the C templates: for instance, if the `#TASK` keyword is used in the C template, it is replaced in the template instance by the name of the task chosen in the GUI.

   Possible developments of new templates call for developments in multiple fields such as neural networks, asynchronous random boolean networks or stochastic cellular automata to mention but a few.

9.5.2 Task compilation

The second tool was motivated by the observation that, during the development of a program, the compilation process is always a tedious and time-consuming task. To reduce this time, the GUI proposes a single button to compile the tasks programs, which calls a compilation script for the task itself. Every time a new task is created, a dedicated compilation script is generated from a script template to include all the requirements of the framework (libraries, compiler flags, etc.) but also so that the user can change, if required, the compilation script of the task.

In its current state, the compilation process can create two types of executable files: a version for a desktop computer that can be used for simulation (see section 9.5.3) or a version for a dedicated hardware platform, such as the ULYSSE processor. The output of the compilation process, reporting potential compilation errors and warnings, is displayed directly in the GUI (shown in the bottom part of Figure 9.3, page 148).

Note that, because the GUI is multi-platform, so are the compilation scripts. For this reason, these scripts are written in a *shell scripting language*, in our case `bash`, that can be run under the major existing operating systems.

9.5.3 Task simulation

The third type of tool is related to the need of checking the functionality of the code early in the design process even though the application is still incomplete.

Because the tasks are independent in stream computing, there exists the opportunity to simulate and debug them separately. Thus, once the task has been written and successfully compiled, the user can easily verify that the newly-created task behaves as predicted by simulating it with proper inputs.

As maintaining two different versions of the code, one aimed at simulation and one for execution, would be a waste of time and resources, we introduced in the templates the possibility to generate small modifications to account for the differences between the code simulated on a PC and on a different platform. Thus, a flag in the template can be set during the compilation process to switch between different `main()` functions. With this flag, even though the code can start in different functions, the computational part of the tasks remains identical for the different versions, the template taking care of calling it according to the situation.

Thanks to the introduction of this mechanism and the execution of the code on a desktop PC, the user can generate, according to his or her preferences, test vectors for the task, assert the code at different locations, etc. This way, checking the task program functionality is simplified, as the output information of the tasks can be directly displayed in the GUI to verify their proper operation.

9.5.4 Task profiling

The fourth tool finds its roots in the fact that, since some tasks may require specific performance constraints, it might prove important to evaluate their most computationally-intensive parts. Thus, we included in the code templates, as well as in the GUI, adequate support to simplify this task.

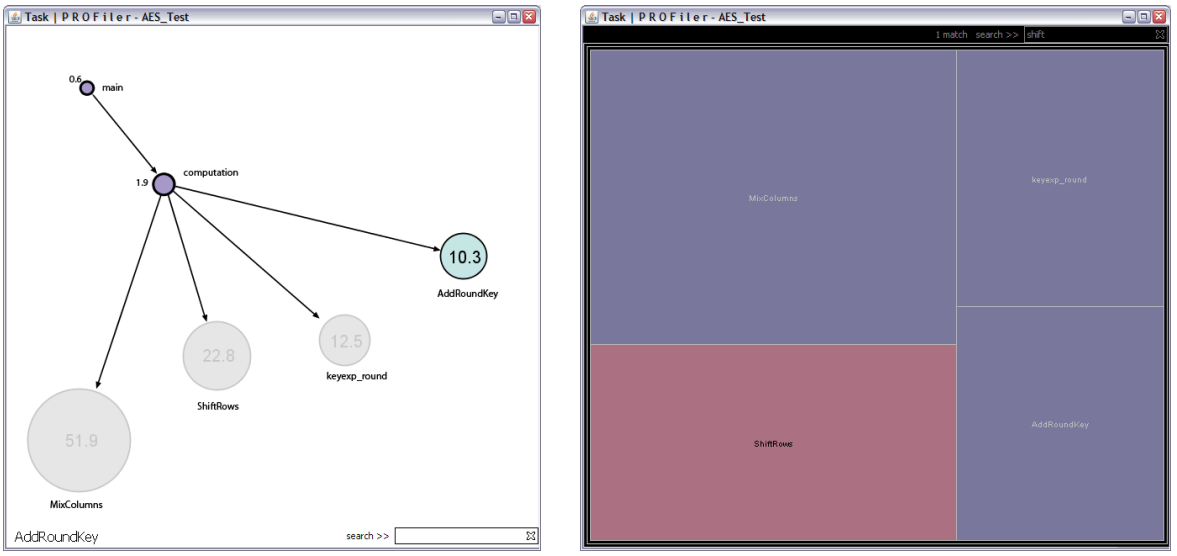
Using the same methods used for task simulation, a template can contain a dedicated function to be filled so that the performance of a task can be estimated. It consists of calling a sufficient number of times the `compute()` function of the task so that the code can be annotated with timing and profiling information.

The annotation itself is performed using a standard profiler, called *gprof*[5] [Graham 82,Graham 04]. To reduce the inaccuracies inherent to the fact that *gprof* is using sampling techniques to measure the time spent in each function, the profiling of the task is done several times before the results are averaged. Thus, the output of this profiler gives us estimates about:

- the total time of execution;
- the number of times each function has been called;

[5]This acronym stands for the *Gnu PROFiler*.

- the percentage of the total running time of the program used by each function;
- the number of seconds accounted for by each function;
- the average number of milliseconds spent in each profiled function per call;
- the average number of milliseconds spent in each profiled function and its descendants per call.



(a) Circular tree graph

(b) Treemap representation

Figure 9.4: Profiling graphs of a task.

Using the code annotated with the profile information, the GUI can then automatically create graphs such as the ones shown in Figure 9.4. This representation of profiling, inspired from tools such as *Valgrind* [Nethercote 04, Nethercote 06], proposes two distinct views to depict in which functions the most time is spent.

The treemap technique [Shneiderman 92], shown on the right of the figure, has the advantage of providing a good overview of the program time repartition whereas the circular tree graph (left side of the figure) presents the call graph[6] of the program, with each function being represented by a circle whose size is proportional to the time spent in the function.

In the example depicted in Figure 9.4, which represents the profile graph of the AES encryption algorithm [Daemen 02], it can be seen that the `MixColumns()` function accounts for roughly 52 percent of the total execution time. This fact is shown by a circle area about two times larger than the `ShiftRows()` circle area (23 percent) in the tree graph. Similarly, the area of the function in the tree map is also two times larger. One can also very rapidly distinguish that the `AddRoundKey()` function is comparable, in terms of execution time, to the `keyexp_round()` function.

One aim of this profiling tool was to provide a *simple and streamlined* way to access performance information so that users can optimize their programs, if required, without needing to know how to set up a software toolchain for that purpose. Moreover, it provides a convenient way to compare two implementations to choose the most performant one.

Another goal of the profiling of task programs is to provide to the software framework an estimation of the complexity of the tasks, an information that will be important in section 9.6.2 where graph execution will be simulated using that timing estimation.

[6]Note that the *call graph*, i.e. the representation of the program as a tree, is generated by the *gprof* tool and extracted to be displayed in the GUI.

If, in our case, that estimation is performed on a desktop PC for simplicity reasons, nothing prevents the implementation of a more complex toolchain to profile the tasks on different hardware setups and under different conditions.

9.5.5 Task library

When the tasks' code has been written, optimized and checked with the aforementioned tools, the tasks are stored in a library that collects every realized task so that each kernel function can be made available in different sections of the GUI program for a quick selection.

This library stores in a central directory all the tasks code and related scripts, which provides a convenient way to retrieve or change the different kernel functions.

9.6 Task graph programming

In the previous section we presented how the development of the kernel functions was simplified within the GUI thanks to a variety of design tools. Following the same methodology, we will examine now how the same ideas can be applied to intuitively build and simulate complete application graphs built with these kernel functions.

For the sake of simplicity, both construction and simulation of graphs are presented within a single window, which allows not only to create graphs but also to explore, in a very intuitive way, how different input conditions influence the dynamic evolution of a graph in simulation. More precisely, the GUI proposes three different tools for graphs:

- *A graph creator*, for editing the graphs.
- *A timing simulator*, which is used to explore the dynamics of graph execution based only on the timing information of the tasks.
- *A parallel simulator*, where the complete graph behavior is simulated with tasks running in parallel on a desktop computer.

Let us now discuss first the *graph creator* module before examining two different simulation modules (section 9.6.2 and 9.6.3).

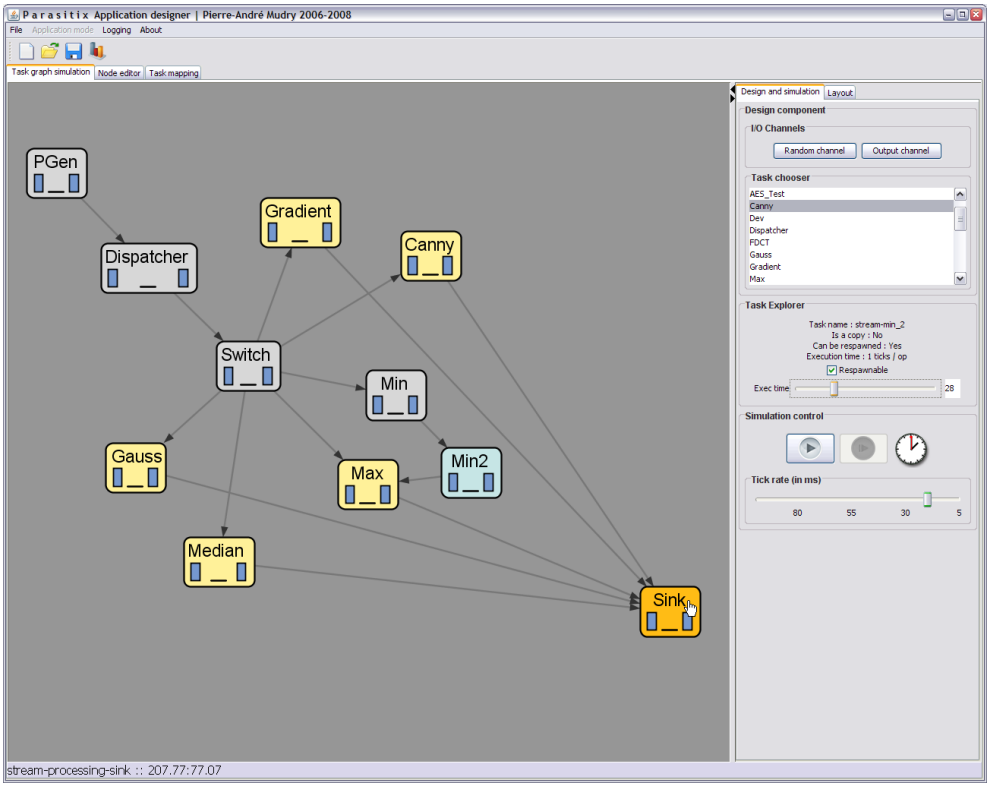
9.6.1 The graph editor

The graph creator and editor module of the GUI is shown in Figure 9.5, the main window containing the graph. This latter can be interactively edited with mouse operations to add and remove vertices, connect them with edges, etc. The different tasks that can be instantiated can be chosen from a list, shown in the top right-end corner of the figure, containing the task library.

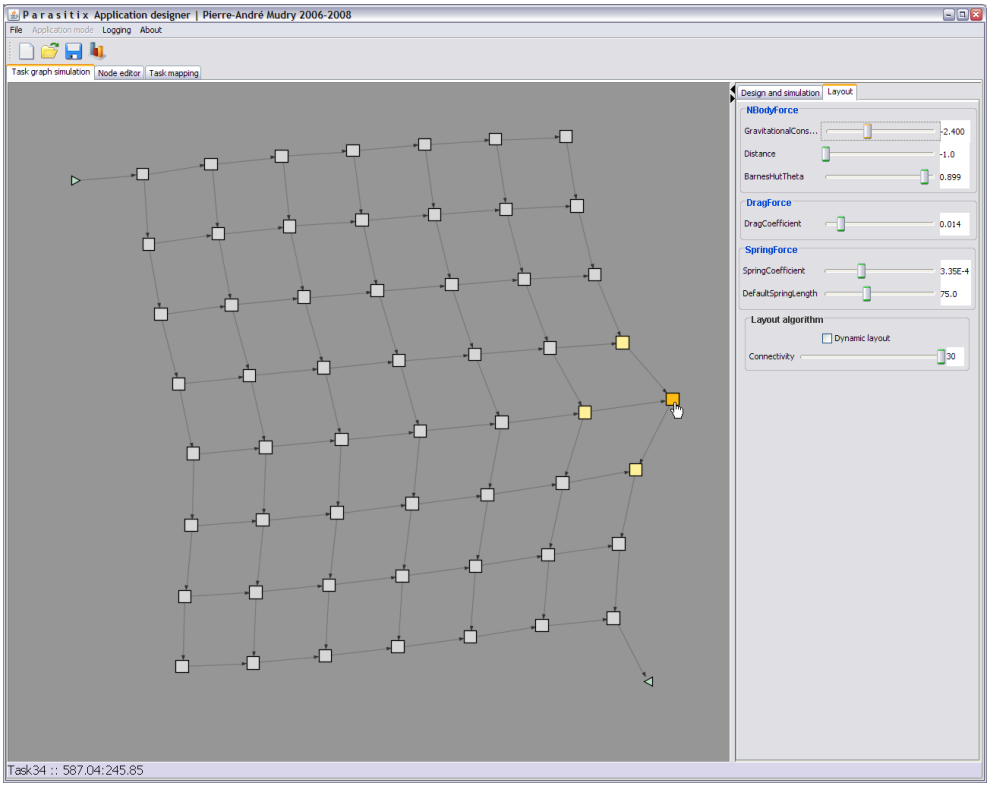
To help the user distinguish the topology of cluttered graphs, hovering the mouse over a task highlights with different colors the nodes that are connected to it (see the *sink* node in Figure 9.5a). For the same reason, different layout algorithms can be applied – statically or dynamically – to the graph so that it is displayed in an aesthetically pleasing way.

For instance, Figure 9.5b shows the application of a dynamic spring layout algorithm that uses a physical model so that the vertices of the graph repel and the edges between the nodes act as springs, a configuration that makes the graph react when nodes are dragged for example. The different configuration parameters of the layout algorithms, such as in this case the default length of the springs, can be seen on the right-hand side of the picture.

Once the editing process is complete, the generated graph can then be saved as a GXL file, a standard file format using the GraphML (XML based) syntax [Brandes 01] for graphs. Using this format has the advantage that the graph can be retrieved in another session but also that it can be used in different graph tools without conversion.



(a) Hovering over a node



(b) Dynamic layout with a spring model

Figure 9.5: The graph editor window.

9.6.2 Timing simulation

The first tool to analyze the dynamics of applications proposes to explore the behavior of the graphs based on the *timing information* of the tasks, which has been extracted during their profiling. This simulation notably enables the user to see the impact of different input scenarios on the graph. These scenarios can be chosen by selecting different input channels, i.e. components that generate different types of data at a certain rate. Thus, different input channels were developed using for example an uniform rate with random data, with data generated following a normal law, etc. More can be implemented by adding appropriate *Java* classes.

Figure 9.6: A timing simulation in the GUI.

Bottleneck identification

The second application of these scenarios in a simulation allows to easily identify bottlenecks in the application. As shown in Figure 9.6, the tasks can be displayed with three vertical bars that represent, from left to right, the input FIFO level of the task, its load[7] and the output FIFO level. In addition, each task can also change the filling color of its background, using different shades to represent the load.

These graphical indications, which are completely flexible because they can be extended to different models, allow to intuitively *identify the presence of "hot-spots"* within the application. Moreover, if realistic scenarios are used, it also becomes possible to refine the splitting of the graph according to that information. Finally, it is also possible to plot in real-time the log of different task parameters and save this information.

[7] As time is discrete within the simulator, we use the notion of "ticks" represent that a time slot has passed. The load of a task is defined as the ratio of non-idle ticks over a given period.

Overall, the animated features of the GUI give a visual feedback to the designer that provides a good insight of the dynamic, global behavior of the application by representing congestions, idleness and other measures with colors and animations. Thus, it becomes possible to visually represent the potential problems of a parallel application in ways that are normally difficult or that can be intuitively misleading.

Limits

One disadvantage of this timing simulation is that, as it relies heavily on estimates, its quality and its match to reality depends on these estimates and on the model. However, because simulation can be performed very quickly but also because the simulation model can be improved to take into account different parameters so that the desired complexity level is attained, this remains a valuable tool that will be reused in the next chapter to examine a task replication algorithm (see chapter 10).

9.6.3 Parallel simulation

If the dynamic simulation proposes to focus on the timing characteristics of the graph, the *parallel simulation module* proposes a way to execute the application code on a distributed environment running on PC before instantiating it on the target hardware.

In this distributed simulation, each task code is executed in a dedicated thread that can communicate with the other tasks as if they were running in the final hardware, with the notable difference that here the FIFO-based communication is simulated transparently in the environment by TCP/IP sockets. Thus, from the tasks' perspective, the code executed is the same as the one on the final hardware, all the functionalities provided by the framework libraries, such as the functions to retrieve the FIFO levels, being also simulated.

With this setup, it is possible to run and verify the application in a controlled execution environment which provides debugging functionalities (consoles, breakpoints and step-by-step execution) that the real hardware can not match in its current state.

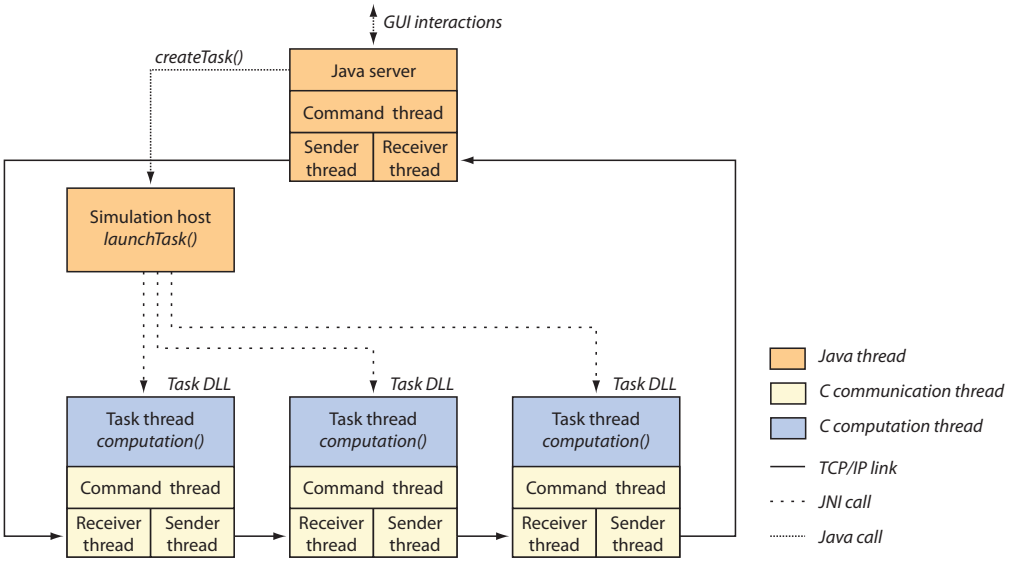


Figure 9.7: The parallel simulation environment showing the simulation of three tasks.

From an implementation point of view, this simulation environment is composed of multiple concurrent *Java* and C components which are presented in Figure 9.7. In this schematic, which represents a graph of three tasks in series (see bottom of the figure), each box containing text represents a distinct

thread running either in C or *Java* and its connections with other boxes depict its interactions with the other components.

Starting from a task graph described in the GUI, it is then possible to effectively execute the application by simulating each task in a parallel thread, also reusing the same C code. A simulation session on a PC follows the following procedure:

- A *Java* server thread is generated to fulfill three different roles:

  1. interact with the GUI (and the user);
  2. ask for tasks to be created or removed;
  3. communicate with the tasks.

  The first role is easily explained by the fact that this server is tightly coupled with the GUI application, the user being able to interact with the simulation in real time, which also explains the second role of this server.

  The communication with the tasks is supported by three different threads, one for receiving application data, one for sending data to the application and a last one for passing command messages to the tasks. This last category includes special messages which can be used to retrieve the status of a task (load, FIFO status, ...) or to dynamically change the topology of the application.

- A *Java* thread (*SimulationHost*), triggered by a task creation request by the server, dynamically loads the tasks' C code (which for this phase is compiled as a dynamically-linked library) using a *Java Native Interface* (JNI) call[8].

- Each time a C task is loaded, it creates two additional threads[9] to simulate the FIFOs present in the hardware interfaces so that the behavior of the task running in a desktop environment is similar to the behavior in hardware. Each of these communication threads use TCP/IP sockets to communicate from task to task, even though solutions other than FIFO communication or TCP/IP could also be implemented if required. However, the flexibility of TCP/IP allows to run the parallel tasks on different machines if necessary. Finally, for each task loaded, a fourth thread is created to react to the commands sent by the *Java* server.

9.6.4 Tasks placement

The placement of the tasks takes as an input the task graph and a processor graph that contains the topology of the target hardware to produce as an output a placement of the tasks onto the processors.

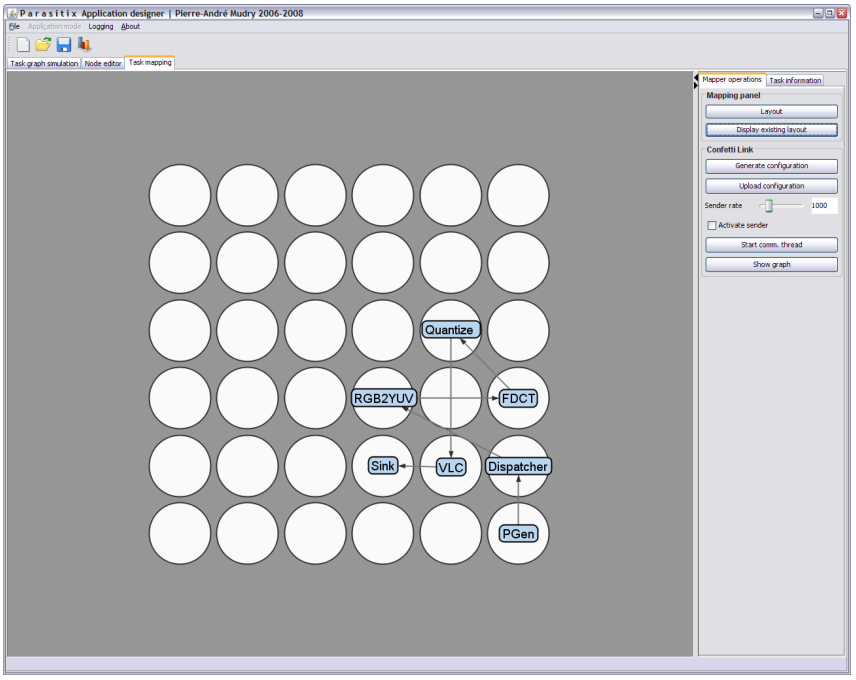
At the time of writing, a single type of processor graph has been written, based on the hardware structure of the CONFETTI platform, even though everything is planned in the software GUI to easily integrate different types of graphs. Also, we only examined with tasks placement the situation in which each processor can run *at most* a single task. The motivation of this decision resides in the lack of a timer interrupt in the ULYSSE processor (see section 2.6.2, page 20) which hinders implementation of a multitasked environment. When used with the CONFETTI platform, the placer module displays (Figure 9.8) the actual topology of the processors as a grid, which helps the user understand the influence of the relative placement of the tasks.

For instance, it is relatively obvious that a placement such as the one presented in Figure 9.8a presents several disadvantages when compared to Figure 9.8b. This fact can be rapidly grasped thanks to graphical hints such as the overlapping edges or the fact that the edges are not horizontal or vertical, which signifies that routing will have to be done through multiple elements.

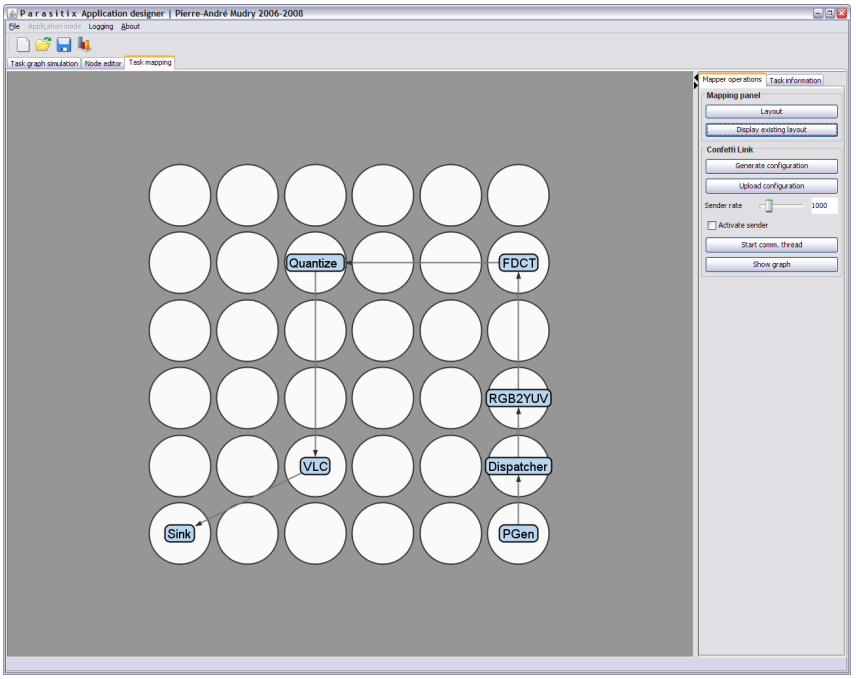
Even if we did not investigate or implement the automatic placement of tasks in this work due to a lack of time, this would be an interesting research opportunity that could leverage on an abundant literature on the subject, such as [Aguilar 97, Bouvry 94].

[8] JNI is used to call C code within a *Java* application, see [Liang 99].

[9] All the C threads are created using POSIX threads so that they are compatible with the Linux operating system as well as with Windows.



(a) Bad placement



(b) Good placement

Figure 9.8: The GUI placement module.

9.6.5 Graph compilation

Once the application graph is complete and the placement of tasks done, the graph can then be *compiled*, i.e. translated into an adequate format to be downloaded to the hardware target. To make this process as extensible as possible, a *shell script* language is used once again. In the case CONFETTI is used as the hardware target for the graph, the procedure contains the following steps:

1. Each task is compiled for the ULYSSE processor, which is done by calling the *makefile* of each task with a flag indicating the executable type.
2. On demand, a USB interface program is called to download the processors' code to the CONFETTI platform according to the mapping information gathered from the placer tool.

The whole procedure of graph compilation is automated and can be launched from the GUI, which makes sense as each alteration of the tasks placement or of the tasks' code require a new graph compilation.

9.6.6 Graph execution

To replicate the presence in CONFETTI of a command channel as used in simulation, it is also possible to send commands to the tasks executed in the real platform thanks to a dedicated program, whose role is basically to translate messages sent with the TCP/IP protocol to USB commands that are then routed to the CONFETTI platform. With this mechanism, the whole hardware platform can be accessed from any device capable of TCP/IP connectivity.

An example of the commands that are implemented and that can be sent or received from CONFETTI – or any other hardware with an adequate protocol – are for instance requesting the status of a particular processor's input or output FIFO, retrieving the load of a processor, requesting a processor replication or migration to another location, etc. In fact, any command can be imagined as their implementation only requires to add functions to the MERCURY API library presented in section 8.5.

9.7 Conclusion

The GUI program that was demonstrated in this chapter constitutes a valuable tool in providing users a system view powerful enough to be able to manage the complexity of parallel systems while remaining simple enough for novices to access complex hardware.

In addition, we also discussed in this chapter how, in order for bio-inspired concepts to be accepted as valuable tools for the design and use of digital computing systems, they will have to be tested and verified on complex real-world applications. The experimental setups associated with research in this field, however, are often of limited use in this context because they remain either too specific or too difficult to use beyond the proof-of-concept stage. By guiding the user through different explicit and intuitive modules in the GUI the process of developing bio-inspired applications such as cellular automata with our approach becomes a matter of hours or minutes.

When compared with a traditional approach, such as the one used for the BioWall where only hardware programmability was possible, this considerable reduction in design time constitutes a major step forward. Moreover, not only can the platform be easily used but also the shift from a complete hardware approach towards a more software-oriented one allows far more flexibility while preserving the capability of accessing the low-level hardware if necessary. A possible target for further research would be to exploit this flexibility to an even greater degree, notably by directly integrating codesign modules for the processor in the GUI, for example to take advantage of novel hardware-software codesign techniques, such as UML, as presented in the work of Bocchio et al. (see, for example, [Bocchio 07, Riccobene 05]).

In the next chapter, we will see how this mixed software/hardware approach is not restricted to bio-inspired computations but can also be used to accelerate standard algorithms, such as AES encryption, thanks to the self-scaling stream processing paradigm.

Bibliography

[Aguilar 97] Jose Aguilar & Erol Gelenbe. *Task assignment and transaction clustering heuristics for distributed systems*. Information Sciences, vol. 97, pages 199–219, 1997.

[Ahmad 99] Ishfaq Ahmad & Yu-Kwong Kwok. *On Parallelizing the Multiprocessor Scheduling Problem*. IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 4, pages 414–432, 1999.

[Bocchio 07] Sara Bocchio, Elvinia Riccobene, Alberto Rosti & Patrizzia Scandurra. *Advances in Design and Specification Languages for Embedded Systems*, chapter "A Model-driven co-design flow for Embedded Systems". Springer, 2007.

[Bouvry 94] Pascal Bouvry, J.-M. Geib & D. Trystram. *Analyse et conception d'algorithmes parallèles*. Hermes, April 1994.

[Brandes 01] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt & M. Scott Marshall. *GraphML Progress Report, Structural Layer Proposal*. In Proceedings of the 9th International Symposium on Graph Drawing (GD'2001), pages 501–512, Heidelberg, 2001. Springer Verlag.

[Buck 04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston & Pat Hanrahan. *Brook for GPUs: stream computing on graphics hardware*. In ACM SIGGRAPH 2004 Papers, pages 777–786. ACM, 2004.

[Daemen 02] Joan Daemen & Vincent Rijmen. *The Design of Rijndael: AES – the Advanced Encryption Standard*. Springer, 2002.

[Dally 03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight & Ujval J. Kapasi. *Merrimac: Supercomputing with Streams*. In SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, page 35, Washington, USA, 2003. IEEE Computer Society.

[Engler 95] D. R. Engler, M. F. Kaashoek & Jr. J. O'Toole. *Exokernel: an operating system architecture for application-level resource management*. SIGOPS Operating Systems Review, vol. 29, no. 5, pages 251–266, 1995.

[Fernando 03] Randima Fernando & Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, Boston, MA, USA, 2003.

[Gajski 82] D. D. Gajski, D. A. Padua, D. J. Kuck & R. H. Kuhn. *A Second Opinion on Data Flow Machines and Languages*. Computer, vol. 15, no. 2, pages 58–69, 1982.

[Garey 79] M. R. Garey & D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W.H. Freeman & Company, January 1979.

[Ghuloum 07] Anwar Ghuloum. *Ct: channelling NeSL and SISAL in C++*. In CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming, pages 1–3, New York, USA, 2007. ACM.

[Gordon 06] Michael I. Gordon, William Thies & Saman Amarasinghe. *Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*. In ASPLOS-XII: Proceedings of the 12th International Conference on Architectural support for programming languages and operating systems, pages 151–162, New York, USA, 2006. ACM.

[Graham 82] Susan L. Graham, Peter B. Kessler & Marshall K. Mckusick. *Gprof: A call graph execution profiler*. In SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, pages 120–126, New York, USA, 1982. ACM.

[Graham 04] Susan L. Graham, Peter B. Kessler & Marshall K. McKusick. *Gprof: A call graph execution profiler*. SIGPLAN Notices, vol. 39, no. 4, pages 49–57, 2004.

[Greensted 07] A.J. Greensted & A.M. Tyrrell. *RISA: A Hardware Platform for Evolutionary Design*. In Proc. IEEE Workshop on Evolvable and Adaptive Hardware (WEAH07), pages 1–7, Honolulu, Hawaii, April 2007.

[Heer 05] Jeffrey Heer, Stuart K. Card & James A. Landay. *Prefuse: a toolkit for interactive information visualization*. In Proceeding of the SIGCHI conference on Human factors in computing systems (CHI '05), pages 421–430, New York, 2005. ACM Press.

[Hutton 07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[Iverson 62] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, 1962.

[Jerraya 05] Ahmed Jerraya & Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, San Francisco, California, 2005.

[Jerraya 06] Ahmed Jerraya, Aimen Bouchhima & Frédéric Pétrot. *Programming models and HW-SW interfaces abstraction for multi-processor SoC*. In Proceedings of the 43rd annual conference on Design automation (DAC'06), pages 280–285, New York, USA, 2006. ACM.

[Johnston 04] Wesley M. Johnston, J. R. Paul Hanna & Richard J. Millar. *Advances in dataflow programming languages*. ACM Computing Survey, vol. 36, no. 1, pages 1–34, 2004.

[Kahn 74] G. Kahn. *The Semantics of a Simple Language for Parallel Programming*. In J. L. Rosenfeld, editor, Proceedings of the IFIP Congress on Information Processing, pages 471–475. North-Holland, New York, USA, 1974.

[Kapasi 02] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens & Brucek Khailany. *The Imagine Stream Processor*. In Proceedings of the IEEE International Conference on Computer Design, pages 282–288, September 2002.

[Lewis 90] T.G. Lewis, H. El-Rewini, J. Chu, P. Fortner & W. Su. *Task Grapher: A Tool for Scheduling Parallel Program Tasks*. Proceedings of the Fifth Distributed Memory Computing Conference, vol. 2, pages 1171–1178, April 1990.

[Liang 99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1999.

[Nesbit 08] K.J. Nesbit, M. Moreto, F.J. Cazorla, A. Ramirez, M. Valero & J.E. Smith. *Multicore Resource Management*. IEEE Micro, vol. 28, no. 3, pages 6–16, May-June 2008.

[Nethercote 04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[Nethercote 06] N. Nethercote, R. Walsh & J. Fitzhardinge. *Building Workload Characterization Tools with Valgrind*. In Proceedingso of the IEEE International Symposium on Workload Characterization, pages 2–2, 2006.

[Peterson 81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Pham 06] D.C Pham, T. Aipperspach & D. Boerstler et al. *Overview of the architecture, circuit design, and physical implementation of a first-generation CELL processor*. IEEE Solid-State Circuits, vol. 41, no. 1, pages 179–196, 2006.

[Riccobene 05] Elvinia Riccobene, Patrizzia Scandurra, Alberto Rosti & Sara Bocchio. *A SoC Design Methodology Involving a UML 2.0 Profile for SystemC*. In Proceedings of the conference on Design, Automation and Test in Europe (DATE'05), pages 704–709, Washington, DC, USA, 2005. IEEE Computer Society.

[Sarkar 89] Vivek Sarkar. *Partioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London and the MIT Press, Cambridge, MA, 1989.

[Shneiderman 92] Ben Shneiderman. *Tree visualization with tree-maps: 2-d space-filling approach*. ACM Transactions on Graphics, vol. 11, no. 1, pages 92–99, 1992.

[Snir 96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker & Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, February 1996.

[Taylor 02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe & Anant Agarwal. *The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs*. IEEE Micro, vol. 22, no. 2, pages 25–35, 2002.

[Tempesti 01] Gianluca Tempesti, Daniel Mange, André Stauffer & Christof Teuscher. *The BioWall: an electronic tissue for prototyping bio-inspired systems*. In Proceedings of the 3rd Nasa/DoD Workshop on Evolvable Hardware, pages 185–192, Long Beach, California, July 2001. IEEE Computer Society.

[Thies 02] William Thies, Michal Karczmarek & Saman P. Amarasinghe. *StreamIt: A Language for Streaming Applications*. In Proceedings of the 11th International Conference on Compiler Construction (CC'02), pages 179–196, London, UK, 2002. Springer-Verlag.

[Thoma 04] Yann Thoma, Gianluca Tempesti, Eduardo Sanchez & J.-M. Moreno Arostegui. *POEtic: An Electronic Tissue for Bio-Inspired Cellular Applications*. BioSystems, vol. 74, pages 191–200, August 2004.

[Upegui 07] Andres Upegui, Yann Thoma, Eduardo Sanchez, Andres Perez-Uribe, Juan-Manuel Moreno Arostegui & Jordi Madrenas. *The Perplexus bio-inspired reconfigurable circuit*. In Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS07), pages 600–605, Washington, USA, 2007.

[van der Wolf 04] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer & Gerben Essink. *Design and programming of embedded multiprocessors: an interface-centric approach*. In Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04), pages 206–217, New York, USA, 2004. ACM.

Chapter 10

Self-scaling stream processing

"The future has already arrived.
It's just not evenly distributed yet."

WILLIAM GIBSON

THE EVER-INCREASING demand in processing power imposed, for example, by the popularity of high-definition media, has revamped interest in multiprocessor systems, which can provide significant advantages in terms of testability or performance scaling; criteria such as time-to-market or lower non-recurrent engineering costs thanks to design reuse are also contributing to this trend. Given the difficulty of devising novel high-performance processor microarchitectures, multiprocessor systems provide another way to exploit the growing number of transistors on a chip by re-using an identical design several times over.

Unfortunately, decades of compiler research have not yet succeeded in providing an automated way to exploit the full potential of parallelism. Consequently, multi-threaded applications are still mostly designed by hand and require strong parallel coding skills, still scarce in the programming community. One solution that has been proposed to speed up the development of such systems is to rely on *pipeline parallelism*, where computation is divided into several stages that process data locally before forwarding them to the next stage. This kind of parallelism, which was introduced in the previous chapter, possesses several advantages, such as good data locality and the fact that, in opposition to task parallelism, data sharing and synchronization are handled in such a way that most of the typical problems of concurrent programming, such as race conditions and deadlocks, are avoided.

10.1 Introduction and motivations

Obviously, several classes of problems solvable in parallel can be tackled more easily by the use of adequate programming paradigms: parallelism at the task level can thus be handled by threads or data-level parallelism by SIMD instructions. However, such paradigms are often of little help when scarce and specific resources have to be used, which is often the case with multiprocessor embedded systems. In this context, which often imposes strict cost and performance requirements, complete hand design of applications remains common. Such a design follows an iterative process: the application is split into different tasks or threads which are then mapped to different processors. The complete system is then profiled to determine if it fulfills all timing requirements and the procedure is repeated until all criteria are met. This time-consuming process has to be repeated for every new application and the quality of the outcome mainly depends on designer experience.

Several efforts to achieve automatic extraction of pipeline parallelism from programs have been attempted, either based on fine-grained (instruction-level) [Ottoni 05] or coarse-grained (function-level) [Dai 05] techniques. Other methods such as code annotation [Thies 07] rely on programmer expertise to extract the underlying parallelism. In addition to pipeline parallelism, task migration in

multiprocessor systems on chip (MPSoC) has been the focus of some interest recently [Bertozzi 06, Ngouanga 06, Pittau 07] and has positioned itself as a partial solution to perform dynamic load distribution in academic and commercial systems.

In this chapter, we aim at demonstrating a new technique that combines the different approaches while offering the system designer an interesting alternative to completely hand-tailored systems in the context of coarse-grained *streaming applications*, a class of problems characterized by a flow of data on which computation is carried out sequentially [Bridges 08, Buck 04].

Our approach leverages the software GUI presented previously and requires the programmer to divide computation into a graph of independent tasks, which are then mapped onto computing nodes. The main novelty of our solution – largely independent of the nature of the processing nodes that can range from the elements of a MPSoC to a grid of computers – resides in the dynamic and distributed execution of an algorithm that enables tasks to *replicate* on demand. This replication can be initiated according to various criteria, such as CPU load or buffer occupation, without requiring the use of a centralized manager, and allows an application to grow and self-organize.

It should be pointed out from the outset that this technique is based on the strong assumption that computing resources are available in large quantities, since it does not hesitate to use resources that are not always strictly necessary. As such, its primary advantage is not optimal performance but the fact that it can used as a dynamic, distributed and scalable load distribution technique which is able not only to quickly respond to workload changes but also to act as a simple resource allocation heuristic in massively parallel embedded systems.

To show the value of the approach, we applied it to a 36-processor system implemented on a scalable mesh of FPGAs for two different case studies: AES encryption and MJPEG compression. In the first case, we obtained up to a ten-fold speedup compared to a static implementation and, in the second case, a throughput multiplication of 11.

This chapter is organized as follows: in the next section, we formulate the problem in terms of a distributed replication algorithm. The following section is dedicated to the design and implementation of the algorithm itself. Thereafter, we present some experimental results which show the efficiency of our approach. Finally, section 10.7 concludes this chapter and introduces future work.

10.2 Self-scaling stream processing

One of the objectives pursued by our software/hardware framework is to provide the user with a simple and automatic method to use parallel systems. As mentioned earlier, pipeline parallelism can be used to increase application performance. Stream processing is an instance of this model in which computation is transformed into a sequence of independent *kernel functions* sequentially applied to a data stream. If these functions can run simultaneously, an execution pipeline is formed.

An important advantage of stream processing is the ease with which it can be adapted to run on highly-parallel computer systems. Thanks to independent kernel functions, concurrency issues are greatly reduced: no data sharing and synchronization are required, except for the passing of stream data between successive pipeline stages. Thus, kernel functions can be separated to run on different, separate computational *nodes* – be they threads, cores, or CPUs – as long as a way to convey data between them exists. This ability to run separately can lead to an enhancement of the static DAG model with the possibility to *dynamically create* and/or *migrate* tasks as we will see now.

10.2.1 Task migration

Task migration consists of changing the mapping of tasks to nodes during the execution of the program. According to [Bouvry 94], this migration can occur in the middle of a computation, an option which requires *checkpointing* mechanisms, or only at certain locations, such as function boundaries. In this work, we will consider this latter option only for the sake of simplicity.

A typical case where migration could occur is when two tasks are executed on a single processor and one of them is migrated to another processor to distribute the load. Another situation that can benefit from task migration is when a different placement of the tasks within the parallel platform

could reduce the distance – hence the communication latency – between two processing nodes. Finally, a third typical situation where task migration could be used is for fault resilience, for example to remove a task from a node experiencing a partial failure.

Generally speaking, the migration process can be undertaken under different conditions that are in most cases determined by performance metrics which are measured centrally. For instance, recently such an approach was chosen and successfully used for task migration in the *Apaches* platform [Saint-Jean 07, Ngouanga 06], a platform that bears many similarities with CONFETTI, notably in terms of resources available. When more imposing systems are considered, many other solutions – MOSIX [Barak 93] or Sprite [Ousterhout 88] to name a few – have been proposed and we refer the interested user to the extensive state-of-the-art review of Milojičić et al. [Milojičić 00] for more insights on process and task migration.

10.2.2 Task duplication

Beyond the speedup obtained from using a distributed pipeline with task migration, it is also possible to increase the throughput of individual pipeline stages by duplicating the same kernel function on different computation nodes and balance the load among them. Once more, this is possible because kernel functions are independent and need not share any state, even between instances of the same function.

The duplication process itself can either be static, i.e. prior to execution, or decided at run-time. In both situations, the question of scheduling posed by duplication is NP-complete [Papadimitriou 88] and heuristics are generally used, such as in [Ahmad 98, Bansal 03, Darbha 98]. In general, the solutions that have been applied (see, for example, Casavant's taxonomy in [Casavant 88]) to the balancing problem aim at maximizing criteria of *efficiency* or *performance*. In the first case, the technique aims at minimizing the resources whilst meeting some given constraints and, in the second case, the technique aims at extracting the most performance at any cost. Some algorithms try to fulfill both criteria, as in [Bozdag 05].

10.2.3 A distributed approach

When considering only the dynamic approach, one disadvantage of most task creation or migration algorithms resides in the presence of a *central manager* that runs an algorithm used to determine the load and balance it among the different processors of the system. Regardless of the kind of multiprocessing environment considered, the use of such a central manager has the advantage that the decision is taken with a complete knowledge of the whole system, but poses a problem in terms of *scalability*. For this reason, we propose a completely distributed approach to share load among processors, which we call *self-scaling stream processing* (SSSP).

In this approach, nodes instantiating kernel functions (hereafter referred to as *tasks*) are able to monitor their own load. When overloaded, a task locates an unused node and replicates its code into it, enlarging its associated pipeline stage and consequently raising the total throughput of the system. The workload – chunks of the processed data stream – is then distributed to the tasks that form a stage in a round-robin manner so that that each task receives an identical load. In short, self-scaling stream processing allows a pipeline to grow and self-organize to balance load between its computational nodes automatically, using local decisions instead of a centralized manager. It has the potential to allow good performance scaling and efficient resource allocation, when used in an appropriate environment.

Let us now examine the different questions that need to be solved to implement this idea:

1. How does a task determines if it is overloaded ?
2. When can/must the replicas be destroyed ?
3. What are the requirements for a task to be easily replicated ?
4. Where does the replica task migrate ?

5. How can the global behavior of the application be maintained with dynamically created and deleted tasks ?

6. How is global performance affected by the dynamic replication of the tasks?

Answering these questions will be the guiding thread of the remainder of this chapter: the first two questions will be examined in detail in section 10.6, the next three in section 10.4 and, finally, the performance question will be discussed with various experiments in section 10.5.

10.3 Design

Preliminary remark *The graphs presented in this section as well as its organization are based on Julien Ruffin's master thesis [Ruffin 08], and were modified with permission.*

With the general aim of the algorithm outlined, we will now focus on the requirements needed to implement the replication algorithm itself.

First, we need a means to distinguish between the different task roles in the pipeline. This is achieved by allocating to a given stage of the pipeline a *group identifier* at initialization. To create the hierarchy present in the pipeline, each cell also receives a list of its successor groups. For each of the successor group ID, the cells keep a list of the members of that group that is updated by notification messages broadcast to the entire grid when a node is created or destroyed.

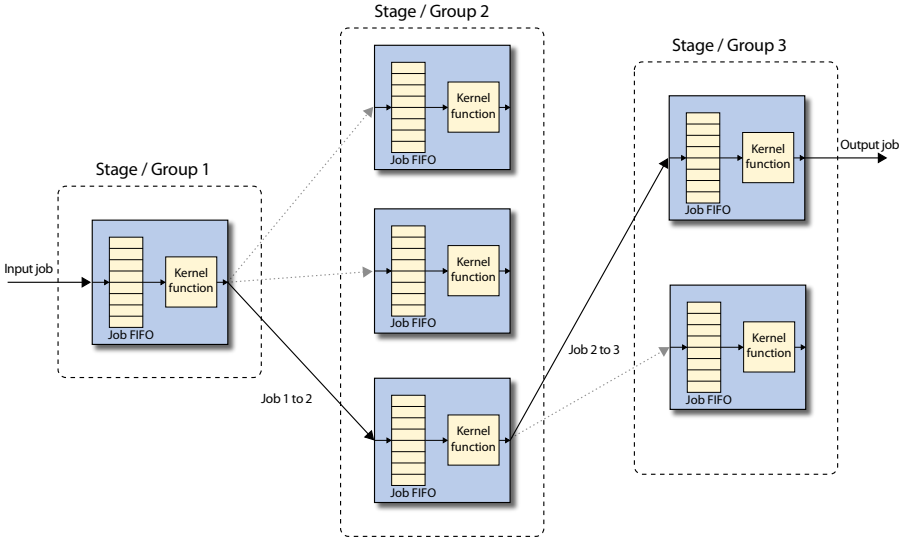


Figure 10.1: Sample 3-stage SSSP pipeline with duplicated stage 2 and 3.

With this organization, every time a cell finishes processing a job, it can dispatch the output further to its successors using a round-robin scheduling technique. For instance, Figure 10.1 shows a three level pipeline with the second stage distributed on three processors and the last stage on two processors. With round-robin scheduling, any incoming packet arriving in stage 1 is forwarded to one of the cells in stage 2 (the exact destination cell changing for each new packet). In the configuration depicted here, the next packet arriving in stage 1 would then be forwarded in the upper cell of stage 2 (which is the next in the schedule) before being sent to a cell in stage 3 that depends upon the round-robin state of the sender cell in stage 2.

10.3.1 Message types and interface

New message types have been added on top of the MERCURY API to account for the needs of this kind of pipeline organization. These messages are independent of the type of MERCURY packet in

which they are encapsulated, i.e. they can be used in broadcast or one-way messages without any difference. Four different message types can be distinguished:

1. *Jobs*. This first message type contains job data that the task has to work on. The header of the message itself contains the sender address, the group ID it is destined to, as well as the payload length of the data.

2. *Control messages*. These messages can be used to request information from the tasks (such as their load) or to inform them of something. The header of the message contains the sender address, a control field that indicates the type of request and an optional payload length.

3. *Status messages*. This type of message is used to reply to a control message. It uses the same type of header as these latter.

4. *Boot message*. Message type received to start the task and also to provide run-time information. For instance, the organization of the pipeline is not fixed at compile time but is dynamic. Thus, the function graph can be constructed at runtime in the system with appropriate messages that indicate to each group its successors, which allows additional flexibility.

10.3.2 Program structure

A typical SSSP application contains several subfunctions that handle the different types of processing required in the task, namely the handling of the data I/O operations and the computation itself.

Cyclic executive

The structure of SSSP programs, shown in Figure 10.2, is based on a cyclic executive program that switches between I/O and computation phases.

A cell must basically execute two types of operations: the first consists in filling the job FIFO with data coming from the network[1] and the second relates to the computing proper.

This kind of setup is relatively common and is generally handled with interrupts triggered by incoming data and an infinite loop to process the jobs available. However, as we mentioned earlier (see section 2.6.2, page 20), interrupts are too complex and computationally intensive in single-instruction architectures and they were not implemented. Consequently, the implementation of the SSSP algorithm within the ULYSSE processor uses polling for I/O operations and static multitasking is obtained with cyclic executives, meaning that the processor always executes a single task, as shown in the following code:

```
void cyclic_executive(){
      while(1){
            fill_queue();
            process_jobs();
      }
}
```

Both functions present in the infinite loop run for a defined period of time in their own processing loop and then return, which enables an alternating behavior. To control the duration of each function, both calls use the timer unit present in ULYSSE to check that a certain amount of time has elapsed.

Of course, the disadvantage of this technique is that the duration of each function has to be determined manually, depending on the ratio of communication time versus computation time of particular applications. As such, the implementation of interrupts would be more elegant but, unfortunately, we did not have time to investigate this solution.

[1] Note that the job FIFO resides in the program memory and is different from the MERCURY FIFO present in hardware.

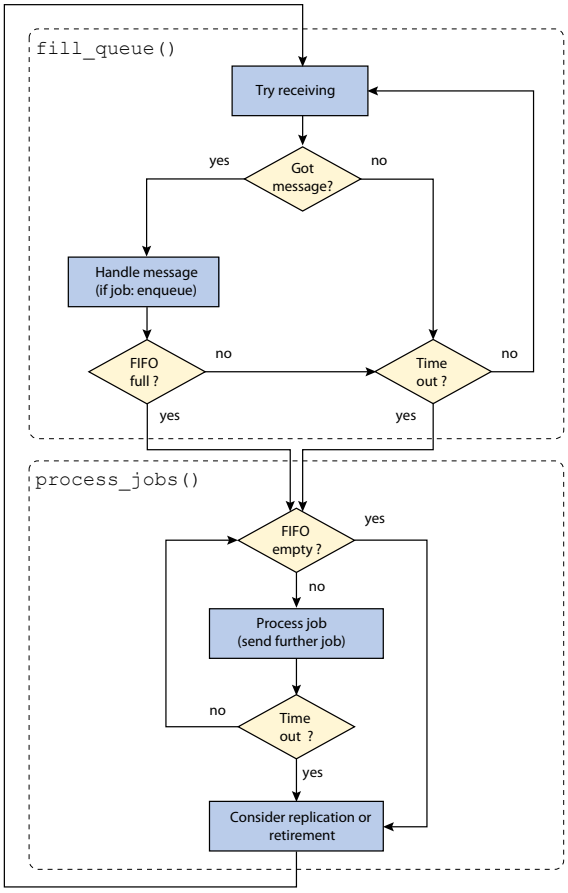


Figure 10.2: Cyclic executive graph.

10.3.3 Programmer interface

The programmer's interface is similar to CAFCA, presented in section 8.6. As a summary of the different functionalities of the SSSP API, the following code depicts the implementation of the identity function, which basically forwards a packet to the next pipeline stage without modifying it:

```
#include "stream-processing-common.h"
const int CODE_BORDER = 0;
#include "stream/stream.h"

void computation(job *j) {
  stream_output(out_job); // Send the job "as-is" to the next stage.
}

void init () {
  app_computation = &computation;
  stream_duplicable = 0; // Set this to set initial dynamic duplicability
}

void main() {
  stream_init(INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y, &init);
  stream_main_loop();
}
```

Listing 10.1: Stream identity function.

As with CAFCA, the SSSP framework contains functions that ought to be called at initialization to locate the user's custom initialization function and the computation function. At startup, a call to the `stream_main_loop()` functions starts the cyclic executive which in turn periodically calls the computation function.

Every kernel computation function must provide a pointer to a `job` structure, for this data type contains not only the sender information but also the data to be processed by the function. The use of a pointer enables to change the size of the data payload along the pipeline if required so that the transmitted packets remain as small as possible.

Once the computation proper is finished, the framework provides the user with a simple function, `stream_output()`, that can be used to forward the packet to the task's successor in the pipeline. It is worth noting here that it is the framework itself, and not the user, that takes care of determining addresses by looking up the target in its next-stage member list using a round-robin selection to balance the load among the successors. If multiple successor groups are present, a similar function exists to choose among them.

10.4 The replication algorithm

The SSSP model specifies that a task should replicate when overloaded to enable the approach described in sections 10.1 and 10.2. In the algorithm considered here, a task simply launches the replication process whenever its input FIFO buffer is filled past a certain threshold. However, the range of possible alternative criteria imply the existence of different *replication policies* which we shall discuss in section 10.6. To avoid overgrowth and allow reuse of computation nodes, SSSP tasks also implement a *retirement policy* similar to the replication policy. Retired tasks stop processing data.

10.4.1 Free cell search algorithm

When replicating, a working SSSP grows by copying its code to *free cells*, computation nodes which are initialized empty at startup or retired SSSP tasks.

To find suitable targets for replication, SSSP tasks in CONFETTI keep track of active computation nodes through messages sent from starting-up and retiring tasks, which are broadcast to the entire processor grid. This "state table" is built dynamically since the very beginning of the system: every time a new cell appears in CONFETTI – either by replication or by instantiation – it broadcasts to the entire grid the fact that it is used. A similar mechanism is also present each time a cell is removed from the grid. Thus, each node knows exactly the state of each cell of the grid (that busy table is also copied in the replication process).

9	10	11	12	13
14	1	2	3	15
16	4		5	17
18	6	7	8	19
20	21	22	23	24

Figure 10.3: Free cell search algorithm order.

When needed, the free cell search algorithm uses this information to find a suitable replication target, beginning with the eight immediate neighbors of the cell it runs in (the exact order[2] is shown in Figure 10.3), then gradually increasing the search perimeter until a free cell is found or it outgrows

[2]Other search orders are possible, for example to reduce the number of hops between two replica.

the bounds of the processor grid. This local search algorithm is based upon the observation that communication is less susceptible to create congestions in the underlying network by having related messages always go to the same area instead of having messages scattered among different locations on the grid.

For very large computational grids, the scalability of the presented technique could pose a problem, notably in terms of memory required to store the list of the available cells. If scale does become an issue (which was not the case in our system), many other techniques can of course be applied. For example, instead of keeping a list to determine the state of a cell, another solution would have been to implement a *query mechanism*. However, concerns about the relative slowness of the approach and the potential issues in terms of concurrency that could appear with such a technique made us prefer the former solution.

10.4.2 The replication function

We will now examine how the replication mechanism is implemented within CONFETTI. As mentioned earlier in section 8.4.1 (page 127), the ULYSSE processor present in the computation nodes in CONFETTI can be reprogrammed at runtime with a specific message. This is very practical for testing, as the PC interface to the system needs only send such a message to the nodes it wants to reprogram with the executable code as payload.

In the case of cell replication, the initiating running node wants to copy its code over to another. For efficiency reasons, no separate copy of the code is kept to this end; the sent code has to be extracted from the node's own. When doing so, it is easy to know where the code begins, as it is loaded from address 0. However, determining where in memory code ends and data begins is non-trivial.

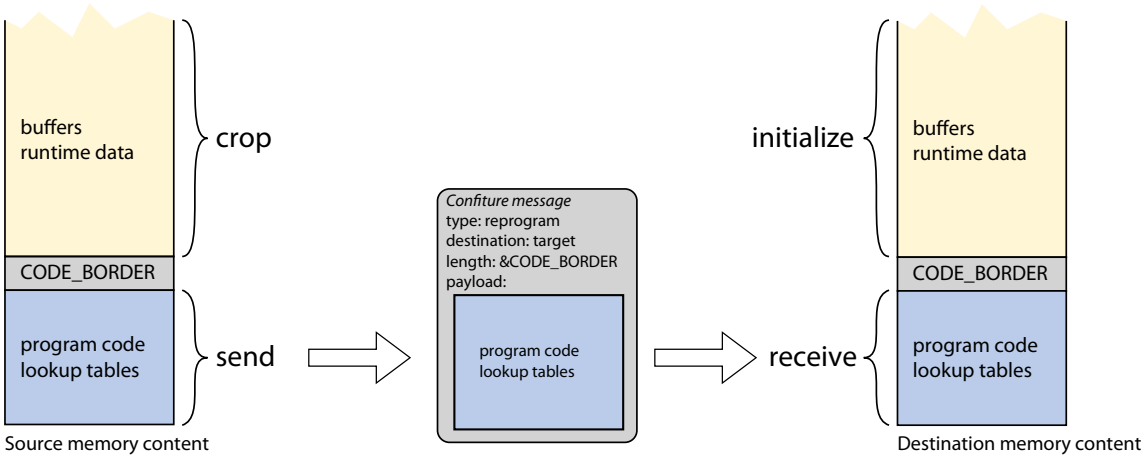


Figure 10.4: Summary of CONFITURE code replication (memory contents not to scale).

Copying the entire memory contents is not an option. Currently, the sent code must hold in a single CONFITURE message, which limits its size to 65534 32-bit words[3]. Bearing this in mind, a straightforward solution would be to write the code size at a given address at compile-time. While possible, this proves relatively inelegant.

Our attempt at a solution uses existing features to make the compiler and assembler work towards easily finding code size. It stems from the following observations:

1. The compiler allocates data words for global variables with initial values *in declaration order* in the output assembly.

[3]The maximum size of a MERCURY packet minus one word used for the CONFITURE header extension.

2. The assembler relocates the aforementioned data words after the code in memory, but before global variables without initial content.

3. Global variables (including buffers) need not be copied as long as they are all initialized at run time. What matters is the room allocated to them by the compiler.

This said, the solution, summarized in Figure 10.4, involves declaring a given global constant with an initial value (currently named `CODE_BORDER`) at the very beginning of the application code (see Listing 10.1 for example). When replicating, the code is taken between address 0 and the address of the constant (obtained in C with the `&CODE_BORDER` expression), then sent. The target's MERCURY receiver checks the message type, as declared in the CONFITURE header, and notices it contains code: its DMA unit writes the message payload from address 0 onwards to replace current application code.

All global variables with initial values declared *before* `CODE_BORDER` will be part of the replicated code. Any global variable declared *after* it will not see its initial content sent. Special care must be taken to ensure such globals are initialized at runtime. Using this setup, a typical SSSP task replication takes between 50 ms and 100 ms.

By modifying the replication function so that it copies its entire memory content, it would also be possible to migrate a task completely. However, it would require to implement a checkpointing mechanism (see [Milojičić 00]) to retrieve the location of the PC after migration, which can be done for example by adding at memory location 0 a jump to the desired PC location.

10.4.3 Startup and replication mechanisms

Now that the question of how replication is performed is answered, let us now examine how it can be used by the SSSP algorithm to copy a cell's code into a free node. In fact, although code replication is fairly simple, precautions need to be taken in order to keep SSSP nodes in a consistent state. In order to understand why, it is necessary to briefly explain the boot process of SSSP tasks.

SSSP nodes start up in two phases. First, they are programmed with the task they must implement. Then, they are set up using a boot message to provide up-to-date runtime information, including pipeline structure and lists of the addresses of the tasks in the next stage of the pipeline. These messages are sent by the PC interface when the SSSP is initialized, or by a replicating task to its target.

The boot message allows major parameters of a SSSP (including pipeline structure) to be changed at initialization without the need to recompile the task code. However, the drawback of this two-phase startup lies in potential concurrency issues when two tasks A and B try to replicate themselves into the same free cell C. The worst-case scenario happens when C gets completely reprogrammed and initialized by A, and then right afterwards by B. In this case, other tasks register C as being a copy of both A and B, which is incorrect.

Since in CONFETTI reprogramming is handled in hardware and immediately suspends currently-running program code to replace it with the new code, a task has no way to indicate others the above-mentioned situation is happening. It is thus an issue fairly specific to our platform.

To avoid this issue of concurrent cell reprogramming, a locking mechanism is used, as depicted in Figure 10.5.

A task wanting to replicate itself first sends a *replication request* message to its target; only when a *replication granted* message has been received does reprogramming take place. This confirmation is not waited for by the originating task, which continues operating normally. Only free cells grant overwriting of their code, *once*. Thus, only the first task from which the request was received will actually send a reprogramming and boot messages. Finally, once a task has been programmed and has received its boot message, it broadcasts its creation to the entire cell grid. This allows all concerned SSSP tasks to update their address lists, making the newly-created task part of the system. From there on, it will receive data to process as a member of the pipeline stage it belongs to.

10.4.4 Limiting growth

The described replication algorithm is susceptible to sudden, explosive growth: once a new task is created, it takes a certain amount of time for the system to stabilize in its new configuration and

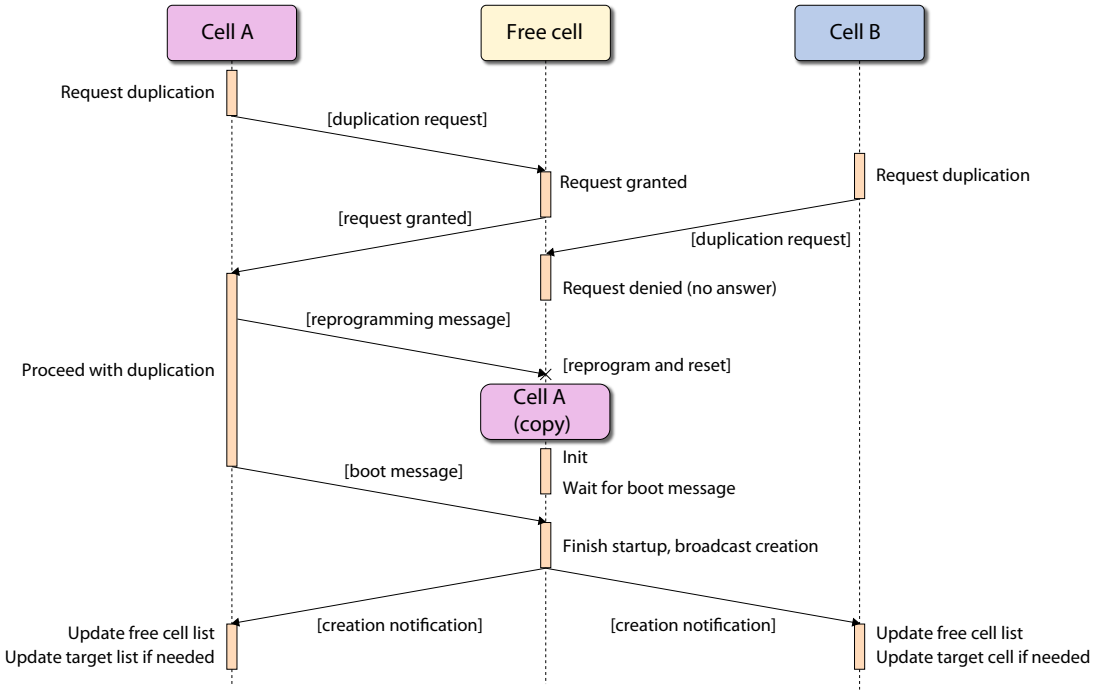


Figure 10.5: Concurrent duplication attempt in SSSP.

balance workload. Before this happens, tasks can still saturate, which can lead to new replications.

To avoid flooding the system, a task that just sent a replication request must wait before sending another. Currently set to one second, this *refractory phase* allows the system to stabilize. Interestingly, this limitation dictates the maximum growth rate of the pipeline, which can potentially double in size every wait period.

10.4.5 Fault tolerance

The free cell search algorithm along with the replication mechanism allow some *fault tolerance* in the nodes themselves, with some similarities with the solution proposed in [Bauch 94]: if during the replication process a faulty node is chosen as a destination, the replication can be retriggered if no answer from the replica is received. The mechanism works as follows: if the destination node can not acknowledge the fact that it is ready as a duplicate, the node that had initiated the copy will choose another destination after a certain time has elapsed.

Another possibility exists to tolerate faults in the system. As each node knows the state of every other cell in the system, a particular node can be marked as non-available, which is achieved through a message broadcast to the system, so that it will not be considered during replication. Thus, it is possible to remove the non-working nodes from the scheduling algorithm dynamically and easily.

10.4.6 Limitations

The scheduling approach we propose has some limitations in its current form, the most important being that it assumes a feed-forward type of application without internal states, at least for the tasks that can replicate. This means that no state can be kept across the computations because that state would not be synchronized among the different replicas. Thus, having neither global state nor centralized control grants the ability to scale to a large number of nodes, but comes at a cost.

Another limitation is that the algorithm, in its current form, does not provide fairness guarantees in resource-limited cases. For instance, when two applications are run simultaneously on the plat-

form, both are considered equally important and the allocation of free processors to one or the other depends only on the timing of the algorithm, free processors being granted to the first asking. However, it would be possible to integrate guarantees, notably by integrating them inside the replication policies and in the replication algorithm so that priorities, either defined manually or automatically could be enforced.

Regarding the fairness of the system, it is also worth noting that, in its current state, the algorithm implies a higher replication probability for the tasks at the beginning of the pipeline than for the tasks at the end. In fact, because the data arrive serially in the pipeline, the tasks getting the data first are more prone to replicate. However, this does not pose any particular problem in the system under the hypothesis that every task that needs replicating has the opportunity to do so.

The last remark on this replication algorithm concerns the granularity of the tasks, which should not be too fine. In fact, tasks must be complex enough that their computation time remains balanced with communication, because running too simple tasks would result in a longer pipeline latency, most of the time being spent waiting for packets. This issue mainly arises because the cyclic executive we use is hand-tuned, which means that the communication delay must be set manually. Again, using interrupts would, at least partially, solve this problem.

10.5 Performance results - case studies

In this section, we present the results we obtained using SSSP on two well-known algorithms, AES encryption and MJPEG encoding, implemented on the CONFETTI system.

10.5.1 Test setup

In order to test the capabilities of our implementation of the SSSP model, we built a test bench on a 6x6-node CONFETTI grid.

Aside from a SSSP pipeline, the test bench contains three tasks needed to use, control and monitor it: *source*, *join* and *split*. The *source* task is in charge of feeding input data to the pipeline input data, as the USB interface to the PC currently cannot provide input fast enough to exploit self-scaling stream processing to its full extent. The *join* task gathers output data from the pipeline, which can come from any task in the last stage. The *split* task acts as the first stage of every pipeline in our test bench: it never duplicates and passes input data to the first computation stage of the pipeline.

Due to network routing constraints, not every node in the grid can send data back to the PC; the *join* task is placed so that it can. In order to obtain the maximum possible throughput between the source and the join, we decided to keep them separate as very simple programs. On the other hand, *split* is a SSSP task which acts as the first stage of every pipeline in our test bench. As the source ignores the fact that tasks can duplicate and feeds inputs to a single fixed address, two possible solutions are either to provide a fixed entry point for the pipeline, or to make the source track the first stage's replications. We decided upon the former because it implies a less complex source.

The *split* task is thus this single entry point: it never duplicates and passes input data to the first "real" computation stage of the pipeline. A potential drawback to having a single entry point is of course the possibility of causing a bottleneck (we discuss the effect of this in our results). However, we can already mention that to limit this bottleneck effect, the *split* task is a stripped-down version of the SSSP base system to render it as fast as possible.

Control of the system is granted through packets sent from the USB interface to the source task to activate or deactivate it as well as to control the interval between two data sends, effectively influencing the rate at which data is fed to the pipeline. Monitoring is handled by the *join* task: it periodically sends a status packet to the PC through the USB interface with performance information gathered internally as well as from updates sent to it by the pipeline and the source task. Processed and logged, these status packets provide the measurements we discuss in the next section.

10.5.2 AES encryption

This simple pipeline is composed – apart from the split and join nodes – of a single node that uses the AES algorithm to encrypt 128-bit data blocks with a 256-bit key. Together, the data and key form the 384-bit packets passed around the system. Using this setup, the maximum sustained throughput of an "empty" pipeline (one with no actual computation) reaches 3930 packets per second. When computation is introduced in form of a non-replicating AES node, the throughput of the pipeline falls to 250 packets encrypted per second.

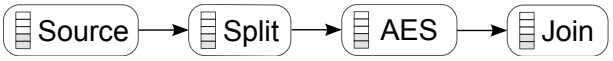


Figure 10.6: AES encryption pipeline.

To observe the effects of dynamic task replication, we use the following workload scenario (*scenario 1*): starting from a very low value, the source's input rate is approximately doubled every 5 seconds until performance peaks. As shown in Figure 10.7, tasks replicate in order to cope with the increasing workload, until the system's performance stabilizes at an average peak rate of 2160 packets per second, which represents a 8.64 increase over the single-node pipeline with a total of 17 processors used. The small fluctuations seen each time the rate is changed can be explained by the fact that, during the replication process, the nodes stop processing data and, when they resume execution, more packets are present in their queues and can be handled directly.

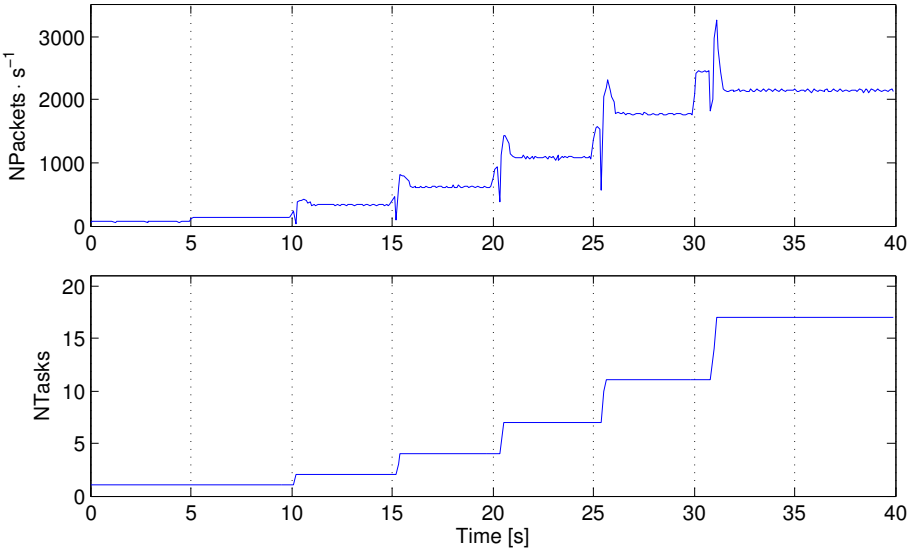


Figure 10.7: AES – Input scenario 1.

Interestingly, the heuristic nature of the replication algorithm yields different results depending on how the input rate changes. For instance, in the second scenario (*scenario 2*) shown in Figure 10.8, instead of a periodic and progressive increase, the input rate is set to its maximum value at time $t = 10$. To cope with this heavy load, 12 additional processors are quickly generated (in about 5 seconds, because of the growth rate limitation mentioned earlier). In this scenario, the system is more efficient and is able to encrypt 2440 packets per second with 13 processors dedicated to AES encryption, achieving a tenfold increase over the single-node pipeline.

To characterize the differences between the two scenarios more precisely, we define the *efficiency* of a task graph as $e = \frac{\alpha}{\beta}$, with α as the number of packets per second processed and β as the number of computation nodes present in the pipeline; it is equivalent to average task throughput.

From the analysis of the two scenarios, we have determined that efficiency depends not only on the input rate but also on its evolution through time. Interestingly, efficiency also depends on the

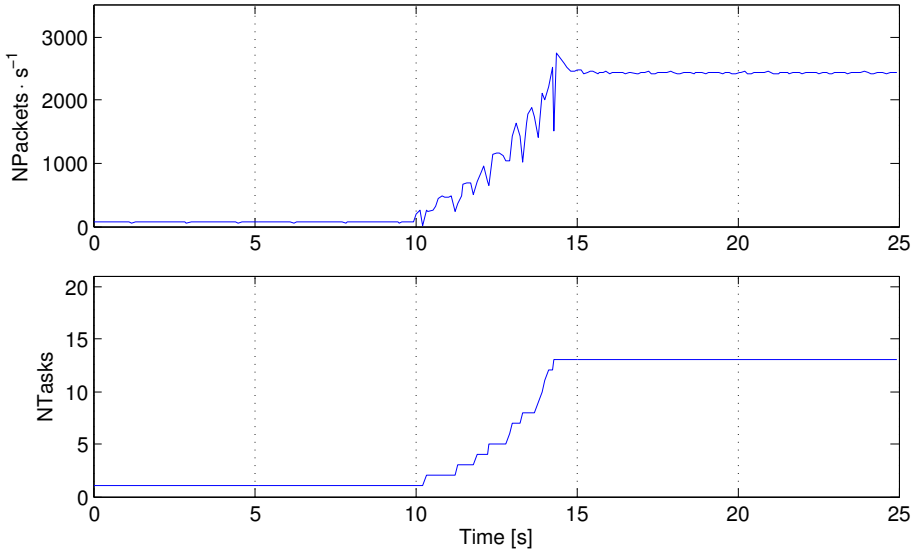


Figure 10.8: AES – Input scenario 2 (best run).

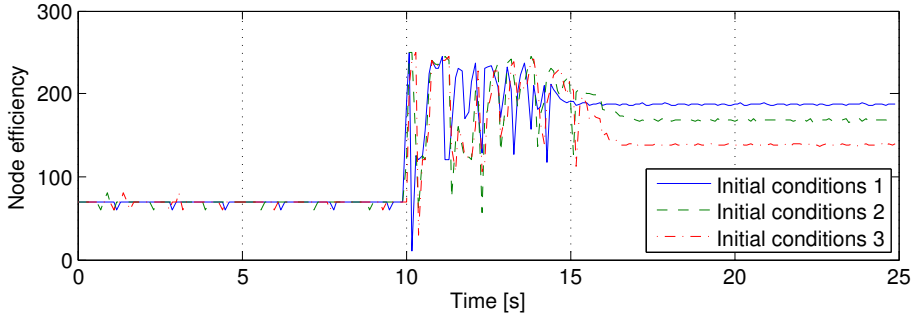


Figure 10.9: Efficiency comparison in scenario 2.

initial placement of tasks on the grid (see section 10.6). This is shown in Figure 10.9, which represents the second scenario with three different initial layouts for the pipeline. Conditions 2 and 3 represent different runs starting with the same initial layout. Placement 1, used in the runs shown up to now, yields more efficient nodes.

10.5.3 MJPEG compression

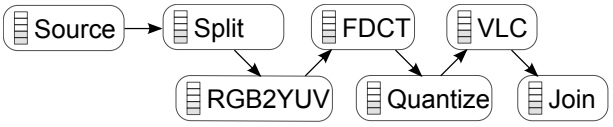


Figure 10.10: MJPEG compression pipeline.

The MJPEG application consists of a four-stage computation pipeline whose last stage, consisting in a variable-length coding (VLC) algorithm, is not allowed to replicate as it keeps history information to improve compression.

The data to be compressed are composed of 192-byte packets corresponding to an 8x8 array of pixels using 24-bit color. With this data type, the maximum rate achievable when the split node's output is fed directly to the join node (bypassing the pipeline) is of 700 packets per second, which roughly

corresponds to 1 $\mathrm{Mbit/s}$. Although this number might seem relatively low, it is worth remembering here that the processing power of the computation nodes is of merely 13 MIPS and that the compiler used, although based on GCC, is still under development and does not currently allow optimizations.

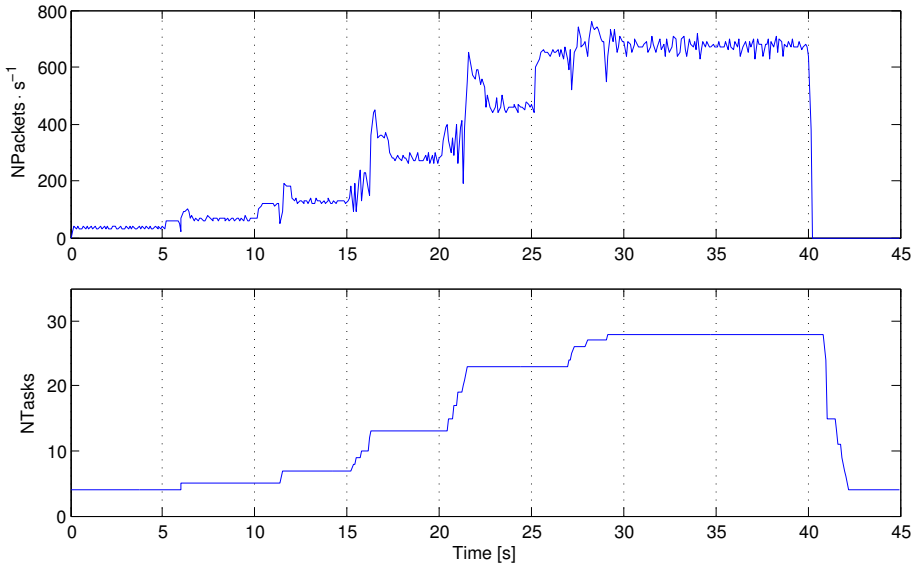


Figure 10.11: MJPEG replication – Scenario 3.

Figure 10.11 depicts a workload scenario in which the source input rate is approximately doubled every 5 seconds. It is similar to scenario 1 in the AES application, with the notable difference that at $t = 40$ the input rate is set to zero. This sudden drop in load results in a fast reduction of the pipeline to its initial state, illustrating the operation of task retirement. When the performance peaks, the average output rate is of 675 packets per second, which is very close to the above-mentioned maximum performance of 700 packets per second. With replication disabled, the performance tops at about 60 packets per second: our technique allows to multiply the throughput by a factor of 11 using 28 processors in total (plus *source*, *split* and *join*).

10.6 Discussion – Policy exploration

Our algorithm demonstrates an interesting degree of scalability. The processors used in our experiment are relatively slow and their use in a normal stream processing pipeline yields modest results. Enabling the pipeline's tasks to replicate results in a considerable increase in maximum throughput, achieved by tapping into available resources as needed, without any need for programmer intervention.

10.6.1 Overall efficiency

Since our algorithm acts as a local heuristic for the task allocation problem, its results vary depending on the pipeline's initial layout, workload, and components. All three have an impact on overall efficiency.

The *initial layout* of the pipeline has an influence on efficiency because of the underlying network. Of the many possible layouts of the initial pipeline (with a single task per stage), some tend to yield increased network congestion, which in turn decreases the average task throughput. One solution to limit the impact of the initial layout resides in the possibility to determine, using the GUI tools presented in the previous chapter, the most computationally intensive tasks so that they are placed in a region where enough room is free to allow replication to happen.

Interestingly, the algorithm tolerates that all processors are allocated: as no further replications are possible, the performance peaks until some cells are made available by the retirement policy.

The *workload* has an influence on efficiency, both by its size and its variations. The larger the workload, the larger the pipeline; but as we showed, the very way the workload is increased can also influence the behavior of the system. A sudden increase to a given workload yields better efficiency than progressively ramping it up: this can be explained by the fact our algorithm has a tendency to overgrow. We already limit the growth rate as explained at the end of section 10.4, and this works well when facing sudden load increases. However, this fails to cope with progressive raises: when the tasks of a stage saturate, due to the round-robin dispatch they will all do so nearly simultaneously and will all want to replicate, making the stage double in size. An overgrown pipeline uses more tasks than necessary and exhibits a lower-than-optimal efficiency.

Finally, the *pipeline's components* also have an influence on the system's behavior, as in the case of MJPEG compression, which consists of four stages instead of one for AES. The MJPEG pipeline has a larger resilience to changes and exhibits less overgrowth than AES, as it consists of more pipeline stages, making it harder to saturate and slower to evolve because in the larger MJPEG pipeline the bottleneck can be found in the last stage, which is the slowest. The early stages provide stability against sudden workload surges that would make this last stage overgrow. This effect can be seen when comparing graphs 10.7 and graphs 10.11: in the latter, it takes longer to attain a stable state.

The reason why efficiency – average task throughput – is stressed as being important here is that it influences throughput. Larger pipelines yield higher network congestion, which means decreased throughput. For this very reason, a pipeline should only consist of the smallest possible number of tasks it needs.

Overall, there are two ways to limit overgrowth, as we will discuss in the next sections.

10.6.2 Overgrowth and replication

The first method is to reduce the effects of abusive replication by introducing a limiting mechanism, for instance by reducing the growth rate. One could imagine modifying our current system by making replication requests be broadcast and have all the tasks of a given pipeline stage reset their limitation timer when they receive such a request, thus reducing the growth rate of the entire stage to approximately one per wait cycle.

10.6.3 Replication policy

The other way is by acting upon its cause: the replication policy. Currently, we use a FIFO occupation policy: when a task's FIFO is filled past a given threshold, the task replicates. This policy has been shown to perform well for threshold values set between 50% and 100% of FIFO size. In our experiments, AES uses a threshold of 16 (full FIFO) while MJPEG has a threshold of 10, except for its last, slower stage which uses a threshold of 8. Through extensive testing, these values were found to maximize throughput.

FIFO-based policies can use more complex metrics than occupation alone and these metrics can be simulated within the GUI environment presented in the previous chapter. Preliminary tests using simulation thus showed that a comparison of the rate at which a FIFO fills versus the rate at which it empties would provide a useful estimate of the workload trend. Time could also be introduced in the decision, to allow replication only if all criteria have been met for a certain period, avoiding overgrowth in the case of transient workload peaks.

Beside FIFO occupation, it would also be possible to base the replication policy on the measured throughput of the task, preferably smoothed over time: a task aware that its throughput is reaching its upper limit could replicate. The difficulty resides in knowing this limit, as it heavily depends on the nature of the computation: it could be provided in advance by measurements on a single cell, or using an adaptive algorithm to determine the value at runtime.

Such a throughput-based policy presents the advantage of being explicitly geared towards maximizing efficiency, since it aims at maximizing task throughput. A drawback is that it assumes maximum task throughput to be more or less constant, i.e. that computation requires roughly the same

time on all input data, which might not always be the case. Moreover, in our experiments, the large influence of network overhead on efficiency and throughput that we observed can be readily explained by the fact that the used processors cannot overlap computation and communication. Then, since network I/O is blocking, congestion slows down communication, which hinders computation and leads to decreased throughput.

10.6.4 Retirement policy

The retirement policy should also be studied. An ideal retirement policy fulfills the same role as an ideal replication policy, namely by maximizing pipeline efficiency. The difference lies in the fact that duplication policies are critical to efficiency when the workload increases, while retirement policies are critical when the workload decreases. Whether a good duplication policy (with the necessary adaptations) is also a good retirement policy is yet to be determined.

The same metrics can be used with replication and retirement policies. We currently use throughput: below a certain average incoming rate, tasks retire. A *grace delay* is used to inhibit retirement for a small amount of time after the task starts, giving it time to ramp up to its normal throughput.

10.6.5 Possible policy improvements

Another improvement to the replication and retirement policies could be the inclusion of additional information. Currently, tasks use information completely local to their processing node as metrics, but it might be possible to take into account more global information such as the relative distances and positions of tasks. Using network load or congestion statistics might also be of great help in making pipelines more efficient. The use of temperature information could allow the pipeline to conform to thermal constraints, of potential interest in embedded systems [Carta 07]. The use of such external metrics could also allow the creation of a free node search algorithm taking into account not only proximity but also pipeline efficiency. In short, letting our heuristic use external information from neighboring tasks and the networking layer could improve its efficiency.

Of course, an effective policy can also be determined by means other than systematic testing. Network calculus, for example, provides a theoretical framework for the study and optimization of networks with packet queues [Boudec 01] that is well adapted to our model. In particular, the extensions of the model proposed by Chakraboty et al. [Chakraborty 03, Chakraborty 06], Thiele et al. with the modular-performance analysis framework[4] (MPA) [Thiele 05] or Maxiaguine [Maxiaguine 04] could enable interesting improvements in the algorithm.

For instance, Maxiaguine in his PhD thesis [Maxiaguine 05] introduces the notion of variability-characterization curves (VCC) which "[...] enable system-level performance analysis of heterogeneous multiprocessor architectures under workloads characterized by variability of several parameters, such as task's execution demands and I/O rates. " [Maxiaguine 05, p. 24]. Implementing such VCC analysis in the global framework would allow an analysis of the tasks' requirements in terms of FIFO memory needed and maximum execution time, which could be useful to provide real-time guarantees and to provide new interesting perspectives for enhancing the policies.

10.7 Conclusion and future work

With the foreseeable development of embedded massively parallel systems in the near future, exciting challenges await the programming community. However, the problem of how to leverage the power of thousands of processors to implement a given application remains unsolved.

In this chapter, we have demonstrated a novel distributed heuristic method to handle dynamic task mapping and load balancing in very large arrays of processors that could be helpful in this context. The SSSP model uses only local information available to the processor and, as such, implies excellent scalability. The validation of the model through the implementation of two standard

[4] `http://www.mpa.ethz.ch/`

applications for embedded systems, AES encryption and MJPEG compression, has shown how this scalability applies to a real system.

The use of a dynamic approach over a static one offers interesting opportunities such as the possibility to add or remove processors as the platform is running or to run multiple applications in parallel on the platform. In addition, the approach is computationally lightweight and largely platform-independent: although our implementation used some of the specific features of the CONFETTI platform (e.g., hardware-managed code replication), none of these features is vital, while on the other hand the lack of more conventional features, such as interrupts, was a distinct disadvantage.

In the future, it would be interesting to implement the SSSP approach on other processor arrays or multiprocessor systems, possibly running real-time operating systems, which could provide different perspectives (and other challenges). Additionally, the study of the algorithm's dynamic behavior along with its applicability in the context of soft-real time systems would provide interesting clues for improvement, most notably when coupled with more complex replication policies including various parameters such as neighborhood status, network load and thermal information. More complex pipeline structures with multiple branches could also be analyzed, for instance in audio data processing.

Another possible improvement would be to consider the application of the SSSP algorithm within an heterogeneous environment, as in [Steensgaard 95], so that the algorithm could harvest the different strengths proposed by the nodes. An example of such a setup would be a platform that includes general purpose processors, DSP and reconfigurable FPGAs. By adding the possibility to the algorithm to know the characteristics of each of these elements, it would be possible to duplicate and/or migrate the task to a more efficient location dynamically. To achieve this behavior, the process state would need to be saved in a machine-independent representation that allows to execute the process on different architectures. This could be achieved, for example, by running virtual machines inside the nodes so that they could execute a hardware-independent language.

Another research opportunity would be not to consider only processing scalability but also to allow some scalability in the networking capabilities of the system to be sure it does not become the bottleneck of the system. Opportunities to attain such a feature reside notably in the reconfigurability of the networking layer of CONFETTI, which could allow for example to perform dynamic modifications of the routing algorithm to adapt to different scenarios at run-time.

Finally, as in CONFETTI each processing unit along with its corresponding routing unit runs in its own clock domain and possesses its own clock generator, it would be possible to explore, in a future work, dynamic clock-scaling algorithms to be used in conjunction with dynamic task replication.

Bibliography

[Ahmad 98] Ishfaq Ahmad & Yu-Kwong Kwok. *On exploiting task duplication in parallel program scheduling*. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 9, pages 872–892, 1998.

[Bansal 03] Savina Bansal, Padam Kumar & Kuldip Singh. *An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems*. IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 6, pages 533–544, 2003.

[Barak 93] Amnon Barak, Shai Guday & Richard G. Wheeler. *The MOSIX distributed operating system*. Springer-Verlag, 1993.

[Bauch 94] A. Bauch, E. Maehle & F. Markus. *A distributed algorithm for fault-tolerant dynamic task scheduling*. In Proceedings of the second Euromicro Workshop on Parallel and Distributed Processing, pages 309–316, 1994.

[Bertozzi 06] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi & Antonio Poggiali. *Supporting task migration in multi-processor systems-on-chip: a feasibility study*. In Proceedings of the conference on Design, automation and test in Europe (DATE'06), pages 15–20, 2006.

[Boudec 01] Jean-Yves Le Boudec & Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, New York, USA, 2001.

[Bouvry 94] Pascal Bouvry, J.-M. Geib & D. Trystram. *Analyse et conception d'algorithmes parallèles*. Hermes, April 1994.

[Bozdag 05] Doruk Bozdag, Fusun Ozguner, Eylem Ekici & Umit Catalyurek. *A Task Duplication Based Scheduling Algorithm Using Partial Schedules*. In Proceedings of the 2005 International Conference on ParallelProcessing (ICPP'05), pages 630–637, Washington, USA, 2005. IEEE Computer Society.

[Bridges 08] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin & David I. August. *Revisiting the Sequential Programming Model for the Multicore Era*. IEEE Micro, January 2008.

[Buck 04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston & Pat Hanrahan. *Brook for GPUs: stream computing on graphics hardware*. In ACM SIGGRAPH 2004 Papers, pages 777–786. ACM, 2004.

[Carta 07] Salvatore Carta, Andrea Acquaviva, Pablo G. Del Valle, David Atienza, Giovanni De Micheli, Fernando Rincon, Luca Benini & Jose M. Mendias. *Multi-processor operating system emulation framework with thermal feedback for systems-on-chip*. In GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI, pages 311–316. ACM, 2007.

[Casavant 88] T.L. Casavant & J.G. Kuhl. *A taxonomy of scheduling in general-purpose distributed computing systems*. IEEE Transactions on Software Engineering, vol. 14, no. 2, pages 141–154, 1988.

[Chakraborty 03] Samarjit Chakraborty, Simon Kunzli & Lothar Thiele. *A General Framework for Analysing System Properties in Platform-Based Embedded System Designs*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), page 10190, Washington, USA, 2003. IEEE Computer Society.

[Chakraborty 06] Samarjit Chakraborty, Yanhong Liu, Nikolay Stoimenov, Lothar Thiele & Ernesto Wandeler. *Interface-Based Rate Analysis of Embedded Systems*. In Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06), pages 25–34, Washington, USA, 2006. IEEE Computer Society.

[Dai 05] Jinquan Dai, Bo Huang, Long Li & Luddy Harrison. *Automatically partitioning packet processing applications for pipelined architectures*. In Proceedings of the 2005 Conference on Programming Language Design and Implementation, pages 237–248, 2005.

[Darbha 98] Sekhar Darbha & Dharma P. Agrawal. *Optimal Scheduling Algorithm for Distributed-Memory Machines*. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 1, pages 87–95, 1998.

[Maxiaguine 04] Alexander Maxiaguine, Simon Künzli & Lothar Thiele. *Workload Characterization Model for Tasks with Variable Execution Demand*. In Proceedings of the conference on Design, automation and test in Europe (DATE'04), page 21040, Washington, USA, 2004. IEEE Computer Society.

[Maxiaguine 05] Alexandre Maxiaguine. *Modeling Multimedia Workloads for Embedded System Design*. PhD thesis, ETH Zurich, Aachen, oct 2005.

[Milojičić 00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler & Songnian Zhou. *Process migration*. ACM Computing Survey, vol. 32, no. 3, pages 241–299, 2000.

[Ngouanga 06] Alex Ngouanga, Gilles Sassatelli, Lionel Torres, Thierry Gil, André Soares & Altamiro Susin. *A contextual resources use: a proof of concept through the APACHES' platform*. In Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pages 44–49, April 2006.

[Ottoni 05] Guilherme Ottoni, Ram Rangan, Adam Stoler & David August. *Automatic Thread Extraction with Decoupled Software Pipelining*. In Proceedings of the 38th International Symposium on Microarchitecture (MICRO 2005), pages 105–118, November 2005.

[Ousterhout 88] John. K. Ousterhout, Andrew R. Cherenson, Frederik Douglis, Michael N. Nelson & Brent B. Welch. *The Sprite Network Operating System*. IEEE Computer, vol. 21, no. 2, pages 23–36, 1988.

[Papadimitriou 88] Christos Papadimitriou & Mihalis Yannakakis. *Towards an architecture-independent analysis of parallel algorithms*. In Proceedings of the 20th annual ACM symposium on Theory of computing (STOC '88), pages 510–513, New York, USA, 1988. ACM.

[Pittau 07] M. Pittau, A. Alimonda, S. Carta & A. Acquaviva. *Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation*. In Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'07), pages 59–64, 2007.

[Ruffin 08] Julien Ruffin. *Self-organized Parallel Computation on the CONFETTI Cellular Architecture*. Master's thesis, École Polytechnique Fédérale de Lausanne, 2008.

[Saint-Jean 07] Nicolas Saint-Jean, Gilles Sassatelli, Pascal Benoit, Lionel Torres & Michel Robert. *HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems*. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07), pages 21–28, 2007.

[Steensgaard 95] B. Steensgaard & E. Jul. *Object and Native Code Thread Mobility*. In Proceedings of the 15th Symposium on Operating Systems principles, pages 68–78, 1995.

[Thiele 05] Lothar Thiele. *Modular Performance Analysis of Distributed Embedded Systems*. In FORMATS 2005, vol. 3829, pages 1–2. Springer Verlag, 2005.

[Thies 07] William Thies, Vikram Chandrasekhar & Saman P. Amarasinghe. *A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs*. In Proceedings of the 40th International Symposium on Microarchitecture (MICRO 2007), pages 356–369, 2007.

Chapter 11

Conclusion

"This is the end,
My only friend, the end."

JIM MORRISSON, *The End*

IN THIS LAST chapter, we will consider to what extent the presented framework fulfills the initial goals that were outlined in the introduction and briefly discuss the outcome of this thesis as whole.

11.1 First objective

The first objective of this thesis was to introduce a processor-grade processing element capable of achieving a flexibility absent in traditional hardware bio-inspired platforms. This task was fulfilled with the introduction of ULYSSE, a complete single-instruction processor that proposes very good flexibility thanks to pluggable functional units. We were able to demonstrate that, despite a non-standard approach to computing, this processor could support the implementation of standard tools, such as a back-end for the GCC compiler, whilst still showing reasonable performance when compared to commercial-grade processors. In addition to a good characterization of its performance, the processor was also validated through different testing methods and tools such as an in-circuit debugger, whose development was simplified thanks to the peculiarities of the architecture itself.

In the context of the design of bio-inspired hardware systems, we showed that it was relatively simple to introduce mechanisms such as genetic evolution, so that it can play an important role in the design. Thus, with the introduction of a novel hardware-software partitioning method, we were able to propose a tool that helps the user determine how to optimize the processor for different applications. This genetic algorithm-based partitioner also showed that, thanks to different and novel hybridization techniques, it was possible to achieve quick and reliable convergence of solutions.

With a processor substantially different from conventional computing units, we were able to bridge the gap between the standard programmable logic on which previous work in bio-inspired hardware focused while preserving enough flexibility to implement real-world applications.

11.2 Second objective

The second objective of this thesis was to propose different hardware and software solutions in the context of cellular computing.

The platform we used for our experiments was CONFETTI, which consists of a scalable three-dimensional array of FPGAs that provides a considerable amount of computational resources and greatly enhances the communication capabilities, compared to traditional solutions. For instance, it allows the implementation of arbitrary connection networks for inter-processor communication, a capability invaluable both to approximate the kind of highly-complex transmission that enables

biological cells to exchange information within an organism and to instantiate self-organization algorithms. We were thus able to develop a complete hardware routing network, based on the HERMES system, modified to work with a limited number of serial links in a GALS environment. In addition, we also developed an USB interface card for the platform that serves both as an interface to a PC and to monitor different parameters of the system.

After implementing the ULYSSE processor inside the CONFETTI nodes and interfacing them to the networking layer with ad-hoc FUs, we developed the CONFITURE library that is used to provide not only messaging capabilities, such as broadcasting, to the processors, but also a way to replicate themselves onto different locations in the system.

This library served as a base to support the CAFCA framework, which was used both to validate the underlying hardware and software setup and to realize synchronous cellular automata on an asynchronous hardware sub-system. Portable, the framework can also be reused very easily on any platform with similar communication capabilities as the ones offered by CONFITURE.

On the software side, in the second part of this thesis we developed a *design flow* that leads from application code, written as tasks graphs realized in C, to a complete parallel system implemented on a hardware substrate. Supported by a GUI that helps create the task graph applications, several tools to program the task were shown, such as template-based programming, that enable for example very quick profiling and testing of the tasks. The GUI also provides different simulation environments for task graphs, one based on a timing characterization of the tasks only while the other provides a complete distributed simulation based on TCP/IP. The GUI also serves as a platform to access the CONFETTI platform to place, download and monitor complete application graphs.

Finally, a distributed and dynamic scheduling algorithm was introduced with the SSSP framework. It allows task graphs to grow and self-organize according to their needs and was validated with two traditional applications used in embedded systems.

11.3 Discussion

Overall, the different hardware modules and software tools constitute a complete, modular codesign framework for cellular architectures. We showed with this thesis that the proposed framework was completely functional and very flexible, as it could be extended at each of its constituting levels: the processor can accommodate additional FUs, the hardware platform can be tailored to different applications by replacing the cells by other elements than FPGAs or by implementing different routing algorithms, and the different software layers can handle applications ranging from cellular automata to more standard applications such as AES. Moreover, an useful feature of the tools is that they are not, for the most part, tied to a hardware implementation: while we used the above-mentioned hardware platform in our experiments, most of the tools are quite general and can be applied to almost any network of computational nodes, whether they be conventional processors or dedicated elements. This flexibility comes from a decoupling between the different hardware and software layers, which allows the programmer to prototype bio-inspired approaches without necessarily knowing hardware description languages and specific implementation details.

The approach we proposed takes advantage of reconfigurability on several levels: at the design level, where the VHDL description of the processors and of the network can be manipulated through a set of more or less automated software tools; at the system level, where processing nodes can be spawned and killed to satisfy the needs of the application or to respond to faults; at the processor level, where the processor can self-reconfigure both its executable code and its very structure (by reconfiguring the FPGA that implements its functional units).

Our final objective was to develop a set of design tools that could allow researchers to exploit this reconfigurability easily and without specific knowledge of the underlying hardware, in order to harness the power of bio-inspired approaches in the context of real-world applications. While this objective was met to some considerable extent, improvements at different levels are possible, as we discussed in the concluding sections of each chapter. Notably, some of the most interesting research directions for future work call for further development of the integration of heterogeneous processing elements in the framework. The introduction of heterogeneity would allow more flexibility in terms

of hardware and performance, at the cost, obviously, of a higher complexity in certain aspects of the system. Some interesting research in terms of software could target the reduction of this cost, notably through the introduction of concepts such as virtualization, that is, mechanisms to separate the operating system from the underlying platform resources. In turn, virtualization, along with other similar mechanisms, could benefit from a more developed version of processor evolution, which could be executed dynamically on the nodes themselves and introduce new hardware functional units as they are required.

Part III

Bibliography and appendices

Complete bibliography

[Abelson 00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman & Ron Weiss. *Amorphous computing*. Communications of the ACM, vol. 43, no. 5, pages 74–82, 2000.

[Agerwala 82] T. Agerwala. *Data Flow Systems: Guest Editors' Introduction*. Computer, vol. 15, no. 2, pages 10–13, 1982.

[Aguilar 97] Jose Aguilar & Erol Gelenbe. *Task assignment and transaction clustering heuristics for distributed systems*. Information Sciences, vol. 97, pages 199–219, 1997.

[Ahmad 98] Ishfaq Ahmad & Yu-Kwong Kwok. *On exploiting task duplication in parallel program scheduling*. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 9, pages 872–892, 1998.

[Ahmad 99] Ishfaq Ahmad & Yu-Kwong Kwok. *On Parallelizing the Multiprocessor Scheduling Problem*. IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 4, pages 414–432, 1999.

[Ahmed 04] Usman Ahmed & Gul N. Khan. *Embedded system partitioning with flexible granularity by using a variant of tabu search*. In Proceedings of the Canadian Conference on Electrical and Computer Engineering, vol. 4, pages 2073–2076, May 2004.

[Al-Dubai 02] A. Y. Al-Dubai, M. Ould-Khaoua & L. M. Mackenzie. *Towards a scalable broadcast in wormhole-switched mesh networks*. In Proceedings of the 2002 ACM symposium on Applied computing (SAC'02), pages 840–844, New York, USA, 2002. ACM.

[Amde 05] M. Amde, T. Felicijan, A. Efthymiou, D. Edwards & L. Lavagno. *Asynchronous On-Chip Networks*. IEE Proceedings Computers and Digital Techniques, vol. 152, no. 2, March 2005.

[Amos 04] Martyn Amos. *Cellular computing*. Oxford University Press, New York, 2004.

[Andriahantenaina 03] A. Andriahantenaina & A. Greiner. *Micro-network for SoC: implementation of a 32-port SPIN network*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), pages 1128–1129, 2003.

[Arnold 01] Marnix Arnold. *Instruction set extension for embedded processors*. PhD thesis, Delft University of Technology, 2001.

[Bansal 03] Savina Bansal, Padam Kumar & Kuldip Singh. *An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems*. IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 6, pages 533–544, 2003.

[Barak 93] Amnon Barak, Shai Guday & Richard G. Wheeler. *The MOSIX distributed operating system*. Springer-Verlag, 1993.

[Baron 08] Max Baron. *VLIW: Failure or Lesson ?* Microprocessor Report, June 2008.

[Bartic 05] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde & R. Lauwereins. *Topology adaptive network-on-chip design and implementation*. IEE Proceedings – Computers and Digital Techniques, vol. 152, no. 4, pages 467–472, 2005.

[Bauch 94] A. Bauch, E. Maehle & F. Markus. *A distributed algorithm for fault-tolerant dynamic task scheduling*. In Proceedings of the second Euromicro Workshop on Parallel and Distributed Processing, pages 309–316, 1994.

[Benini 01] Luca Benini & Giovanni de Micheli. *Power networks on chips: energy-efficient and reliable interconnect design for SoCs*. In Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01), pages 33–38, October 2001.

[Benini 02] Luca Benini & Giovanni de Micheli. *Networks on Chips: A New SoC Paradigm*. Computer, vol. 35, no. 1, pages 70–78, 2002.

[Bertozzi 06] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi & Antonio Poggiali. *Supporting task migration in multi-processor systems-on-chip: a feasibility study*. In Proceedings of the conference on Design, automation and test in Europe (DATE'06), pages 15–20, 2006.

[Bishop 99] Benjamin Bishop, Thomas P. Kelliher & Mary Jane Irwin. *A detailed analysis of MediaBench*. In Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'99), pages 448–455, 1999.

[Bjerregaard 06] Tobias Bjerregaard & Shankar Mahadevan. *A survey of research and practices of Network-on-chip*. ACM Computing Survey, vol. 38, no. 1, page 1, 2006.

[Bocchio 07] Sara Bocchio, Elvinia Riccobene, Alberto Rosti & Patrizzia Scandurra. *Advances in Design and Specification Languages for Embedded Systems*, chapter "A Model-driven co-design flow for Embedded Systems". Springer, 2007.

[Boudec 01] Jean-Yves Le Boudec & Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, New York, USA, 2001.

[Boule 05] Marc Boule & Zeljko Zilic. *Incorporating Efficient Assertion Checkers into Hardware Emulation*. In Proceedings of the 2005 International Conference on ComputerDesign (ICCD'05), pages 221–228, Washington, USA, 2005. IEEE Computer Society.

[Bouvry 94] Pascal Bouvry, J.-M. Geib & D. Trystram. *Analyse et conception d'algorithmes parallèles*. Hermes, April 1994.

[Bozdag 05] Doruk Bozdag, Fusun Ozguner, Eylem Ekici & Umit Catalyurek. *A Task Duplication Based Scheduling Algorithm Using Partial Schedules*. In Proceedings of the 2005 International Conference on ParallelProcessing (ICPP'05), pages 630–637, Washington, USA, 2005. IEEE Computer Society.

[Brandes 01] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt & M. Scott Marshall. *GraphML Progress Report, Structural Layer Proposal*. In Proceedings of the 9th International Symposium on Graph Drawing (GD'2001), pages 501–512, Heidelberg, 2001. Springer Verlag.

[Bridges 08] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin & David I. August. *Revisiting the Sequential Programming Model for the Multicore Era*. IEEE Micro, January 2008.

[Buck 04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston & Pat Hanrahan. *Brook for GPUs: stream computing on graphics hardware*. In ACM SIGGRAPH 2004 Papers, pages 777–786. ACM, 2004.

[Campi 07] Fabio Campi, Antonio Deledda, Matteo Pizzotti, Luca Ciccarelli, Pierluigi Rolandi, Claudio Mucci, Andrea Lodi, Arseni Vitkovski & Luca Vanzolini. *A dynamically adaptive DSP for heterogeneous reconfigurable platforms*. In Proceedings of the conference on Design, automation and test in Europe (DATE'07), pages 9–14, San Jose, CA, USA, 2007. EDA Consortium.

[Carta 07] Salvatore Carta, Andrea Acquaviva, Pablo G. Del Valle, David Atienza, Giovanni De Micheli, Fernando Rincon, Luca Benini & Jose M. Mendias. *Multi-processor operating system emulation framework with thermal feedback for systems-on-chip*. In GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI, pages 311–316. ACM, 2007.

[Casavant 88] T.L. Casavant & J.G. Kuhl. *A taxonomy of scheduling in general-purpose distributed computing systems*. IEEE Transactions on Software Engineering, vol. 14, no. 2, pages 141–154, 1988.

[Catania 97] Vincenzo Catania, Michele Malgeri & Marco Russo. *Applying Fuzzy Logic to Codesign Partitioning*. IEEE Micro, vol. 17, no. 3, pages 62–70, 1997.

[Chakraborty 03] Samarjit Chakraborty, Simon Kunzli & Lothar Thiele. *A General Framework for Analysing System Properties in Platform-Based Embedded System Designs*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), page 10190, Washington, USA, 2003. IEEE Computer Society.

[Chakraborty 06] Samarjit Chakraborty, Yanhong Liu, Nikolay Stoimenov, Lothar Thiele & Ernesto Wandeler. *Interface-Based Rate Analysis of Embedded Systems*. In Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06), pages 25–34, Washington, USA, 2006. IEEE Computer Society.

[Chang 91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter & Wen mei W. Hwu. *IMPACT: an architectural framework for multiple-instruction-issue processors*. SIGARCH Computer Architecture News, vol. 19, no. 3, pages 266–275, 1991.

[Chappell 02] Stephen Chappell & Chris Sullivan. *Handel-C for Co-Processing and Co-Design of Field Programmable System on Chip*. In Proceedings of the Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA02), pages 65–70, 2002.

[Cocke 90] John Cocke & Victoria Markstein. *The evolution of RISC technology at IBM*. IBM Journal of Research and Development, vol. 34, no. 1, pages 4–11, 1990.

[Colwell 88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth & Paul K. Rodman. *A VLIW architecture for a trace scheduling compiler*. IEEE Transactions on Computers, vol. 37, no. 8, pages 967–979, 1988.

[Coppola 04] Marcello Coppola, Stephane Curaba, Miltos D. Grammatikakis, Riccardo Locatelli, Giuseppe Maruccia & Francesco Papariello. *OCCN: a NoC modeling framework for design exploration*. Journal of Systems Architecture, vol. 50, no. 2-3, pages 129–163, 2004.

[Corporaal 91] Henk Corporaal & Hans (J.M.) Mulder. *MOVE: a framework for high-performance processor design*. In Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pages 692–701, New York, USA, 1991. ACM.

[Corporaal 93] Henk Corporaal & Paul van der Arend. *MOVE32INT, a sea of gates realization of a high performance transport triggered architecture*. Microprocessing and Microprogramming, vol. 38, pages 53–60, 1993.

[Corporaal 97] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, USA, 1997.

[Corporaal 99] Henk Corporaal. *TTAs: missing the ILP complexity wall*. Journal of Systems Architecture, vol. 45, pages 949–973, 1999.

[Cruz 00] José-Lorenzo Cruz, Antonio González, Mateo Valero & Nigel P. Topham. *Multiple-banked register file architectures*. SIGARCH Computer Architecture News, vol. 28, no. 2, pages 316–325, 2000.

[Daemen 02] Joan Daemen & Vincent Rijmen. *The Design of Rijndael: AES – the Advanced Encryption Standard*. Springer, 2002.

[Dai 05] Jinquan Dai, Bo Huang, Long Li & Luddy Harrison. *Automatically partitioning packet processing applications for pipelined architectures*. In Proceedings of the 2005 Conference on Programming Language Design and Implementation, pages 237–248, 2005.

[Dally 01] William J. Dally & Brian Towles. *Route packets, not wires: on-chip inteconnection networks*. In DAC '01: Proc. 38th Conf. on Design automation, pages 684–689, New York, USA, 2001. ACM Press.

[Dally 03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight & Ujval J. Kapasi. *Merrimac: Supercomputing with Streams*. In SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, page 35, Washington, USA, 2003. IEEE Computer Society.

[Darbha 98] Sekhar Darbha & Dharma P. Agrawal. *Optimal Scheduling Algorithm for Distributed-Memory Machines*. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 1, pages 87–95, 1998.

[Day 83] John D. Day & Hubert Zimmermann. *The OSI reference model*. Proceedings of the IEEE, vol. 71, no. 12, pages 1334–1340, 1983.

[de Micheli 06] Giovanni de Micheli & Luca Benini. *Networks on Chips: Technology and Tools (Systems on Silicon)*. Morgan Kaufmann, first edition, 2006.

[Dennis 88] J. B. Dennis & G. R. Gao. *An efficient pipelined dataflow processor architecture*. In Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Deshpande 07] Sameera Deshpande & Uday P. Khedker. *Incremental Machine Description for GCC*. Indian Institute of Technology, Bombay, 2007. `http://www.cse.iitb.ac.in/~uday/soft-copies/incrementalMD.pdf`.

[Dick 98] Robert P. Dick & Niraj K. Jha. *MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pages 920–935, October 1998.

[Duato 02] Jose Duato, Sudhakar Yalamanchili & Lionel Ni. *Interconnection Networks: An Engineering Approach*. Elsevier Science, revised edition, 2002.

[Dumitraş 03] Tudor Dumitraş, Sam Kerner & Radu Mărculescu. *Towards on-chip fault-tolerant communication*. In ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation, pages 225–232, New York, USA, 2003. ACM.

[Dutta 00] Sandeep Dutta. *Anatomy of a Compiler – A Retargetable ANSI-C Compiler*. Circuit Cellar Ink, no. 121, pages 30–35, Aug. 2000.

[EEM 08] EEMBC. *DENBench 1.0 – Software benchmark databook*, 2008. `http://www.eembc.org/techlit/datasheets/denbench_db.pdf`.

[Eisner 06] Cindy Eisner & Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Eles 97] Petru Eles, Krysztof Kuchcinski, Zebo Peng & Alexa Doboli. *System Level Hardware/-Software Partioning Based on Simulated Annealing and Tabu Search*. Design Automation for Embedded Systems, vol. 2, pages 5–32, 1997.

[Engler 95] D. R. Engler, M. F. Kaashoek & Jr. J. O'Toole. *Exokernel: an operating system architecture for application-level resource management*. SIGOPS Operating Systems Review, vol. 29, no. 5, pages 251–266, 1995.

[Epalza 04] Marc Epalza, Paolo Ienne & Daniel Mlynek. *Dynamic reallocation of functional units in superscalar processors*. In Proceedings of the 9th Asia-Pacific Computer Systems Architecture Conference, pages 185–198, Beijing, August 2004.

[Ernst 93] Rolf Ernst, Jörg Henkel & Thomas Benner. *Hardware-Software Cosynthesis for Microcontrollers*. In IEEE Design & Test of Computers, pages 64–75, December 1993.

[Ernst 02] Rolf Ernst, Jörg Henkel & Thomas Benner. *Readings in Hardware/Software Co-Design*, chapter Hardware-software cosynthesis for microcontrollers, pages 18–29. Kluwer Academic Publishers, 2002.

[Fernando 03] Randima Fernando & Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, Boston, MA, USA, 2003.

[Flynn 97] D. Flynn. *AMBA: enabling reusable on-chip designs*. IEEE Micro, vol. 17, no. 4, pages 20–27, Jul/Aug 1997.

[Flynn 99] Michael J. Flynn, Patrick Hung & Kevin W. Rudd. *Deep-Submicron Microprocessor Design Issues*. IEEE Micro, vol. 19, no. 4, pages 11–22, 1999.

[Foster 03] Harry Foster, David Lacey & Adam Krolnik. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Foster 05] Harry Foster, E. Marschner & D. Shoham. *Introduction to PSL*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Fraser 91] C. W. Fraser & D. R. Hanson. *A Retargetable Compiler for ANSI C*. Technical report CS–TR–303–91, Princeton University, Princeton, N.J., 1991.

[Gajski 82] D. D. Gajski, D. A. Padua, D. J. Kuck & R. H. Kuhn. *A Second Opinion on Data Flow Machines and Languages*. Computer, vol. 15, no. 2, pages 58–69, 1982.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1995.

[Ganguin 08] Michel Ganguin. *A GCC Backend for the MOVE Architecture*. Master's thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2008.

[Gardner 85] Martin Gardner. *Wheels, Life, and Other Mathematical Amusements*. W.H. Freeman & Company, 1985.

[Garey 79] M. R. Garey & D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W.H. Freeman & Company, January 1979.

[Ghuloum 07] Anwar Ghuloum. *Ct: channelling NeSL and SISAL in C++*. In CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming, pages 1–3, New York, USA, 2007. ACM.

[Gordon 06] Michael I. Gordon, William Thies & Saman Amarasinghe. *Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*. In ASPLOS-XII: Proceedings of the 12th International Conference on Architectural support for programming languages and operating systems, pages 151–162, New York, USA, 2006. ACM.

[Gough 04] Brian J. Gough & Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[Govil 99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang & Mendel Rosenblum. *Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors*. In SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles, pages 154–169, New York, USA, 1999. ACM Press.

[Grafe 89] V. G. Grafe, G. S. Davidson, J. E. Hoch & V. P. Holmes. *The Epsilon dataflow processor*. SIGARCH Computer Architecture News, vol. 17, no. 3, pages 36–45, 1989.

[Graham 82] Susan L. Graham, Peter B. Kessler & Marshall K. Mckusick. *Gprof: A call graph execution profiler*. In SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, pages 120–126, New York, USA, 1982. ACM.

[Graham 04] Susan L. Graham, Peter B. Kessler & Marshall K. McKusick. *Gprof: A call graph execution profiler*. SIGPLAN Notices, vol. 39, no. 4, pages 49–57, 2004.

[Greensted 07] A.J. Greensted & A.M. Tyrrell. *RISA: A Hardware Platform for Evolutionary Design*. In Proc. IEEE Workshop on Evolvable and Adaptive Hardware (WEAH07), pages 1–7, Honolulu, Hawaii, April 2007.

[Guerrier 00] P. Guerrier & A. Greiner. *A generic architecture for on-chip packet-switched interconnections*. In Proceedings of the conference on Design, automation and test in Europe (DATE'00), pages 250–256, 2000.

[Gupta 92] Rajesh K. Gupta & Giovanni de Micheli. *System-level Synthesis using Re-programmable Components*. In Proceedings of the European Design Automation Conference (EDAC), pages 2–7, August 1992.

[Halfhill 08] Tom R. Halfhill. *Intel's Tiny Atom – New low-power microarchitecture rejuvenates the embedded x86*. Microprocessor Report, vol. 1, pages 1–13, April 2008.

[Harkin 01] J. Harkin, T. M. McGinnity & L.P. Maguire. *Genetic algorithm driven hardware-software partitioning for dynamically reconfigurable embedded systems*. Microprocessors and Microsystems, vol. 25, no. 5, pages 263–274, August 2001.

[Hartenstein 99] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann & Ulrich Nageldinger. *Mapping Applications onto Reconfigurable Kress Arrays*. In FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, pages 385–390, London, UK, 1999. Springer-Verlag.

[Heer 05] Jeffrey Heer, Stuart K. Card & James A. Landay. *Prefuse: a toolkit for interactive information visualization*. In Proceeding of the SIGCHI conference on Human factors in computing systems (CHI '05), pages 421–430, New York, 2005. ACM Press.

[Heikkinen 02] J. Heikkinen, J. Takala, A. G. M. Cilio & H. Corporaal. *On efficiency of transport triggered architectures in DSP applications*, pages 25–29. N. Mastorakis, January 2002.

[Heikkinen 05] J. Heikkinen, A. Cilio, J. Takala & H. Corporaal. *Dictionary-based program compression on transport triggered architectures*. In IEEE International Symposium on Circuits and Systems (ISCAS'2005), vol. 2, pages 1122–1125, May 2005.

[Henkel 98] Jörg Henkel & Rolf Ernst. *High-Level Estimation Techniques for Usage in Hardware/Software Co-Design*. In Asia and South Pacific Design Automation Conference, pages 353–360, 1998.

[Henkel 01] Jörg Henkel & Rolf Ernst. *An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity that is Driven by High-Level Estimation Techniques*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 2, pages 273–289, April 2001.

[Hennessy 03] John L. Hennessy & David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[Hoffmann 04] Ralph Hoffmann. *Processeur à haut degré de parallélisme basé sur des composantes sérielles*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[Hoogerbrugge 94] Jan Hoogerbrugge & Henk Corporaal. *Transport-Triggering vs. Operation-Triggering*. In Proceedings of the 5th International Conference Compiler Construction, pages 435–449, January 1994.

[Hoogerbrugge 96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[Hou 96] Junwei Hou & Wayne Wolf. *Process Partitioning for Distributed Embedded Systems*. In Proceedings of the 4th International Workshop on Hardware/Software Co-Design (CODES'96), page 70, Washington, USA, 1996. IEEE Computer Society.

[Huck 00] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder & Rumi Zahir. *Introducing the IA-64 Architecture*. IEEE Micro, vol. 20, no. 5, pages 12–23, 2000.

[Hutton 07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[Iverson 62] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, 1962.

[Jääskeläinen 07] Pekka Jääskeläinen, Vladimír Guzma, Andrea Cilio, Teemu Pitkänen & Jarmo Takala. *Codesign toolset for application-specific instruction-set processors*. In Reiner Creutzburg, Jarmo Takala & Jianfei Cai, editors, Proceedings SPIE Multimedia on Mobile Devices 2007, vol. 6507. SPIE, February 2007.

[Jain 04] Diviya Jain, Anshul Kumar, Laura Pozzi & Paolo Ienne. *Automatically customising VLIW architectures with coarse grained application-specific functional units*. In Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04), pages 17–32, Amsterdam, September 2004.

[Jerraya 05] Ahmed Jerraya & Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, San Francisco, California, 2005.

[Jerraya 06] Ahmed Jerraya, Aimen Bouchhima & Frédéric Pétrot. *Programming models and HW-SW interfaces abstraction for multi-processor SoC*. In Proceedings of the 43rd annual conference on Design automation (DAC'06), pages 280–285, New York, USA, 2006. ACM.

[Johnson 91] William M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[Johnston 04] Wesley M. Johnston, J. R. Paul Hanna & Richard J. Millar. *Advances in dataflow programming languages*. ACM Computing Survey, vol. 36, no. 1, pages 1–34, 2004.

[Kahn 74] G. Kahn. *The Semantics of a Simple Language for Parallel Programming*. In J. L. Rosenfeld, editor, Proceedings of the IFIP Congress on Information Processing, pages 471–475. North-Holland, New York, USA, 1974.

[Kanter 06] David Kanter. *EEMBC Energizes Benchmarking*. Microprocessor Report, vol. 1, pages 1–7, July 2006.

[Kapasi 02] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens & Brucek Khailany. *The Imagine Stream Processor*. In Proceedings of the IEEE International Conference on Computer Design, pages 282–288, September 2002.

[Karim 02] F. Karim, A. Nguyen & S. Dey. *An interconnect architecture for networking systems on chips*. IEEE Micro, vol. 22, no. 5, pages 36–45, 2002.

[Koza 92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Kumar 02] S. Kumar, A. Jantsch, J.P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja & A. Hemani. *A network on chip architecture and design methodology*. In IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), pages 105–112, April 2002.

[Lee 97] Chunho Lee, Miodrag Potkonjak & William H. Mangione-Smith. *MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems*. In MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, pages 330–335, Washington, USA, 1997. IEEE Computer Society.

[Levy 05] Markus Levy. *Evaluating Digital Entertainment System Performance*. Computer, vol. 38, no. 7, pages 68–72, 2005.

[Lewis 90] T.G. Lewis, H. El-Rewini, J. Chu, P. Fortner & W. Su. *Task Grapher: A Tool for Scheduling Parallel Program Tasks*. Proceedings of the Fifth Distributed Memory Computing Conference, vol. 2, pages 1171–1178, April 1990.

[Liang 99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1999.

[Liang 00] Jian Liang, Sriram Swaminathan & Russell Tessier. *aSOC: A Scalable, Single-Chip Communications Architecture*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, vol. 0, pages 37–46, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[Lin 04] C.H. Lin, Y. Xie & W. Wolf. *LZW-based Code Compression for VLIW Embedded Systems*. In Proceedings of the conference on Design, automation and test in Europe (DATE'04), pages 76–81, 2004.

[Lipovski 75] G. Jack Lipovski & J.A. Anderson. *A Micronetwork for Resource Sharing*. In Microarchitecture of Computer Systems, page 223, Nice, France, 1975.

[Lipovski 77] G. Jack Lipovski. *On virtual memories and micronetworks*. SIGARCH Computer Architecture News, vol. 5, no. 7, pages 125–134, 1977.

[López-Vallejo 03] Marisa López-Vallejo & Juan Carlos López. *On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques*. ACM Transactions on Design Automation of Electronic Systems, vol. 8, no. 3, July 2003.

[Lozano 95] Luis A. Lozano & Guang R. Gao. *Exploiting short-lived variables in superscalar processors*. In Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28), pages 292–302, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[Lysecky 05] Roman Lysecky & Frank Vahid. *A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning*. In Proceedings of the conference on Design, automation and test in Europe (DATE'05), pages 18–23, Washington, USA, 2005. IEEE Computer Society.

[Marescaux 02] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde & Rudy Lauwereins. *Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs*. In Proceedings of 12th International Conference on Field-Programmable Logic and Applications (FPL '02) - The Reconfigurable Computing Is Going Mainstream, pages 795–805, London, UK, 2002. Springer-Verlag.

[Marescaux 03] Théodore Marescaux, Jean-Yves Mignolet, Andrei Bartic, W. Moffat, Diederik Verkest, Serge Vernalde & Rudy Lauwereins. *Networks on Chip as Hardware Components of an OS for Reconfigurable Systems*. In Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL'03), pages 595–605, 2003.

[Martin 97] Milo M. Martin, Amir Roth & Charles N. Fischer. *Exploiting dead value information*. In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30), pages 125–135, Washington, USA, 1997. IEEE Computer Society.

[Maxiaguine 04] Alexander Maxiaguine, Simon Künzli & Lothar Thiele. *Workload Characterization Model for Tasks with Variable Execution Demand*. In Proceedings of the conference on Design, automation and test in Europe (DATE'04), page 21040, Washington, USA, 2004. IEEE Computer Society.

[Maxiaguine 05] Alexandre Maxiaguine. *Modeling Multimedia Workloads for Embedded System Design*. PhD thesis, ETH Zurich, Aachen, oct 2005.

[Maxim 04] Maxim. *Introduction to the MAXQ Architecture, Application Note 3222*. `http://pdfserv.maxim-ic.com/en/an/AN3222.pdf`, 2004.

[McNairy 03] Cameron McNairy & Don Soltis. *Itanium 2 Processor Microarchitecture*. IEEE Micro, vol. 23, no. 2, pages 44–55, 2003.

[Mello 05] Aline Mello, Leonel Tedesco, Ney Calazans & Fernando Moraes. *Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC*. In Proceedings of the 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), pages 178–183. ACM, 2005.

[Mello 06] Aline Mello, Leonel Tedesco, Ney Calazans & Fernando Moraes. *Evaluation of current QoS Mechanisms in Networks on Chip*. In Proceedings of the International Symposium on System-on-Chip, pages 1–4, November 2006.

[Meszaros 03] G. Meszaros, S. M. Smith & J. Andrea. *The Test Automation Manifesto*, vol. 2753 of *Lecture Notes in Computer Science*, pages 73–81. Springer Berlin / Heidelberg, September 2003.

[Millberg 04] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar & Axel Jantsch. *The Nostrum Backbone - a Communication Protocol Stack for Networks on Chip*. In Proceedings of the 17th International Conference on VLSI Design (VLSID'04), page 693, Washington, USA, 2004. IEEE Computer Society.

[Milojičić 00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler & Songnian Zhou. *Process migration*. ACM Computing Survey, vol. 32, no. 3, pages 241–299, 2000.

[Mitchell 98] Melanie Mitchell. *Computation in Cellular Automata: A Selected Review*. In Nonstandard Computation, pages 95–140. VCH Verlagsgesellschaft, 1998.

[Moraes 04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller & Luciano Ost. *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. Integrated VLSI Journal, vol. 38, no. 1, pages 69–93, 2004.

[Mucci 08] Claudio Mucci, Luca Vanzolini, Ilario Mirimin, Daniele Gazzola, Antonio Deledda, Sebastian Goller, Joachim Knaeblein, Axel Schneider, Luca Ciccarelli & Fabio Campi. *Implementation of parallel LFSR-based applications on an adaptive DSP featuring a pipelined configurable Gate Array*. In Proceedings of the conference on Design, automation and test in Europe (DATE'08), pages 1444–1449, New York, USA, 2008. ACM.

[Murali 06] Srinivasan Murali, David Atienza, Luca Benini & Giovanni de Micheli. *A Multi-Path Routing Strategy with Guaranteed In-order Packet Delivery and Fault Tolerance for Networks on Chips*. In Proceedings of the Design Automation Conference (DAC'06), pages 845–848, 2006.

[Murali 07] Srinivasan Murali, David Atienza, Luca Benini & Giovanni de Micheli. *A method for routing packets across multiple paths in NoCs with in-order delivery and fault-tolerance guarantees*. VLSI Design, vol. 2007, page 11, 2007.

[Myers 81] Glenford J. Myers. *Advances in Computer Architectures*. Wiley-Interscience, 1981.

[Nesbit 08] K.J. Nesbit, M. Moreto, F.J. Cazorla, A. Ramirez, M. Valero & J.E. Smith. *Multicore Resource Management*. IEEE Micro, vol. 28, no. 3, pages 6–16, May-June 2008.

[Nethercote 04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[Nethercote 06] N. Nethercote, R. Walsh & J. Fitzhardinge. *Building Workload Characterization Tools with Valgrind*. In Proceedingso of the IEEE International Symposium on Workload Characterization, pages 2–2, 2006.

[Ngouanga 06] Alex Ngouanga, Gilles Sassatelli, Lionel Torres, Thierry Gil, André Soares & Altamiro Susin. *A contextual resources use: a proof of concept through the APACHES' platform*. In Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pages 44–49, April 2006.

[Ni 93] Lionel M. Ni & Philip K. McKinley. *A Survey of Wormhole Routing Techniques in Direct Networks*. IEEE Computer, vol. 26, no. 2, pages 62–76, 1993.

[Nikhil 89] R. S. Nikhil. *Can dataflow subsume von Neumann computing?* SIGARCH Computer Architecture News, vol. 17, no. 3, pages 262–272, 1989.

[Ottoni 05] Guilherme Ottoni, Ram Rangan, Adam Stoler & David August. *Automatic Thread Extraction with Decoupled Software Pipelining*. In Proceedings of the 38th International Symposium on Microarchitecture (MICRO 2005), pages 105–118, November 2005.

[Oudghiri 92] H. Oudghiri & B. Kaminska. *Global weighted scheduling and allocation algorithms*. In Proceedings of the European Conference on Design Automation (DATE'92), pages 491–495, March 1992.

[Ousterhout 88] John. K. Ousterhout, Andrew R. Cherenson, Frederik Douglis, Michael N. Nelson & Brent B. Welch. *The Sprite Network Operating System*. IEEE Computer, vol. 21, no. 2, pages 23–36, 1988.

[Palsberg 98] Jens Palsberg & C. Barry Jay. *The Essence of the Visitor Pattern*. In Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC, pages 9–15, 1998.

[Pande 05] Partha Pratim Pande, Cristian Grecu, André Ivanov, Resve Saleh & Giovanni de Micheli. *Design, Synthesis, and Test of Network on Chips*. IEEE Design & Test of Computers, vol. 22, no. 5, pages 404–413, 2005.

[Papadimitriou 88] Christos Papadimitriou & Mihalis Yannakakis. *Towards an architecture-independent analysis of parallel algorithms*. In Proceedings of the 20th annual ACM symposium on Theory of computing (STOC '88), pages 510–513, New York, USA, 1988. ACM.

[Patterson 85] David A. Patterson. *Reduced instruction set computers*. Communications of the ACM, vol. 28, no. 1, pages 8–21, January 1985.

[Patterson 98] David A. Patterson & John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, second edition, 1998.

[Pellerin 05] David Pellerin & Edward A. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005.

[Peterson 81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Petkov 92] N. Petkov. *Systolic Parallel Processing*. Elsevier Science Inc., New York, USA, 1992.

[Pham 05] D.C Pham, E. Behnen, M. Bolliger, H.P. Hostee, C. Johns, J. Kalhe, A. Kameyama & J. Keaty. *The design methodology and implementation of a first-generation CELL processor: a multi-core SoC*. In Proceedings of the Custom Integrated Circuits Conference, pages 45–49. IEEE Computer Society, September 2005.

[Pham 06] D.C Pham, T. Aipperspach & D. Boerstler et al. *Overview of the architecture, circuit design, and physical implementation of a first-generation CELL processor*. IEEE Solid-State Circuits, vol. 41, no. 1, pages 179–196, 2006.

[Pitkänen 05] Teemu Pitkänen, Tommi Rantanen, Andrea G. M. Cilio & Jarmo Takala. *Hardware Cost Estimation for Application-Specific Processor Design*. In Embedded Computer Systems: Architectures, Modeling, and Simulation, vol. 3553 of *Lecture Notes in Computer Science*, pages 212–221. Springer Berlin / Heidelberg, 2005.

[Pittau 07] M. Pittau, A. Alimonda, S. Carta & A. Acquaviva. *Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation*. In Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'07), pages 59–64, 2007.

[Rabaey 98] J. Rabaey & M. Wan. *An energy-conscious exploration methodology for reconfigurable DSPs*. In Proceedings of the conference on Design, automation and test in Europe (DATE'98), pages 341–342, Washington, USA, 1998. IEEE Computer Society.

[Rau 89] B. Ramakrishna Rau, David W.L. Yen & Ross A. Towle. *The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs*. IEEE Computer, vol. 22, no. 1, pages 12–35, 1989.

[Renders 94] J.-M. Renders & H. Bersini. *Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways*. In Proceedings of the First IEEE Conference on Evolutionary Computation, vol. 1, pages 312–317, June 1994.

[Riccobene 05] Elvinia Riccobene, Patrizzia Scandurra, Alberto Rosti & Sara Bocchio. *A SoC Design Methodology Involving a UML 2.0 Profile for SystemC*. In Proceedings of the conference on Design, Automation and Test in Europe (DATE'05), pages 704–709, Washington, DC, USA, 2005. IEEE Computer Society.

[Rijpkema 01] Edwin Rijpkema, Kees Goossens & Paul Wielage. *A Router Architecture for Networks on Silicon*. In Proceedings of the 2nd Workshop on Embedded Systems (Progress'01), pages 181–188, 2001.

[Rossier 08] Joël Rossier. *Self-Replication of Complex Digital Circuits in Programmable Logic Devices*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2008.

[Ruffin 08] Julien Ruffin. *Self-organized Parallel Computation on the CONFETTI Cellular Architecture*. Master's thesis, École Polytechnique Fédérale de Lausanne, 2008.

[Saint-Jean 07] Nicolas Saint-Jean, Gilles Sassatelli, Pascal Benoit, Lionel Torres & Michel Robert. *HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems*. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07), pages 21–28, 2007.

[Sanchez 96] Eduardo Sanchez, Daniel Mange, Moshe Sipper, Marco Tomassini, Andrés Pérez-Uribe & André Stauffer. *Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware*. In Proceedings of the First International Conference on Evolvable Systems (ICES'96), pages 35–54, London, UK, 1996. Springer-Verlag.

[Sarkar 89] Vivek Sarkar. *Partioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London and the MIT Press, Cambridge, MA, 1989.

[Schlansker 00] Michael S. Schlansker & B. Ramakrishna Rau. *EPIC: Explicitly Parallel Instruction Computing*. Computer, vol. 33, no. 2, pages 37–45, 2000.

[Shneiderman 92] Ben Shneiderman. *Tree visualization with tree-maps: 2-d space-filling approach*. ACM Transactions on Graphics, vol. 11, no. 1, pages 92–99, 1992.

[Sipper 99] Moshe Sipper. *The emergence of cellular computing*. Computer, vol. 32, no. 7, pages 18–26, July 1999.

[Sipper 00] Moshe Sipper & Eduardo Sanchez. *Configurable chips meld software and hardware*. Computer, vol. 33, no. 1, pages 120–121, January 2000.

[Snir 96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker & Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, February 1996.

[Srinivasan 98] V. Srinivasan, S. Radhakrishnan & R. Vemuri. *Hardware/software partitioning with integrated hardware design space exploration*. In Proceedings of the conference on Design, automation and test in Europe (DATE'98), pages 28–35, Washington, USA, 1998. IEEE Computer Society.

[Stallman 07] Richard M. Stallman & The GCC Developer Community. *GNU Compiler Collection Internals. For GCC version 4.3.0*. Free Software Foundation, 2007. `http://gcc.gnu.org/onlinedocs/gcc.pdf`.

[Steensgaard 95] B. Steensgaard & E. Jul. *Object and Native Code Thread Mobility*. In Proceedings of the 15th Symposium on Operating Systems principles, pages 68–78, 1995.

[Succhi 99] Giancarlo Succhi & Raymond W. Wong. *The Application of JavaCC to Develop a C/C++ Preprocessor*. ACM SIGAPP Applied Computing Review, vol. 7, no. 3, 1999.

[Tabak 80] Daniel Tabak & G. Jack Lipovski. *MOVE Architecture in Digital Controllers*. IEEE Transactions on Computers, vol. 29, no. 2, pages 180–190, 1980.

[Taylor 02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe & Anant Agarwal. *The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs*. IEEE Micro, vol. 22, no. 2, pages 25–35, 2002.

[Teehan 07] Paul Teehan, Mark Greenstreet & Guy Lemieux. *A Survey and Taxonomy of GALS Design Styles*. IEEE Design and Test, vol. 24, no. 5, pages 418–428, 2007.

[Tempesti 01] Gianluca Tempesti, Daniel Mange, André Stauffer & Christof Teuscher. *The BioWall: an electronic tissue for prototyping bio-inspired systems*. In Proceedings of the 3rd Nasa/DoD Workshop on Evolvable Hardware, pages 185–192, Long Beach, California, July 2001. IEEE Computer Society.

[Tempesti 03] Gianluca Tempesti & Christof Teuscher. *Biology Goes Digital: An array of 5,700 Spartan FPGAs brings the BioWall to "life"*. XCell Journal, pages 40–45, Fall 2003.

[Teuscher 03] Christof Teuscher, Daniel Mange, André Stauffer & Gianluca Tempesti. *Bio-Inspired Computing Tissues: Towards Machines that Evolve, Grow, and Learn*. BioSystems, vol. 68, no. 2–3, pages 235–244, February–March 2003.

[Thiele 05] Lothar Thiele. *Modular Performance Analysis of Distributed Embedded Systems*. In FORMATS 2005, vol. 3829, pages 1–2. Springer Verlag, 2005.

[Thies 02] William Thies, Michal Karczmarek & Saman P. Amarasinghe. *StreamIt: A Language for Streaming Applications*. In Proceedings of the 11th International Conference on Compiler Construction (CC'02), pages 179–196, London, UK, 2002. Springer-Verlag.

[Thies 07] William Thies, Vikram Chandrasekhar & Saman P. Amarasinghe. *A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs*. In Proceedings of the 40th International Symposium on Microarchitecture (MICRO 2007), pages 356–369, 2007.

[Thoma 04] Yann Thoma, Gianluca Tempesti, Eduardo Sanchez & J.-M. Moreno Arostegui. *POETIC: An Electronic Tissue for Bio-Inspired Cellular Applications*. BioSystems, vol. 74, pages 191–200, August 2004.

[Thoma 05] Yann Thoma. *Tissu Numérique Cellulaire à Routage et Configuration Dynamiques*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[Upegui 07] Andres Upegui, Yann Thoma, Eduardo Sanchez, Andres Perez-Uribe, Juan-Manuel Moreno Arostegui & Jordi Madrenas. *The Perplexus bio-inspired reconfigurable circuit*. In Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS07), pages 600–605, Washington, USA, 2007.

[Vahid 94] Frank Vahid, Daniel D. Gajski & Jie Gong. *A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning*. In Proceedings of the conference on European design automation (EURO-DAC'94), pages 214–219, 1994.

[Vahid 95] Frank Vahid & Daniel D. Gajski. *Incremental Hardware Estimation During Hardware/-Software Functional Partitioning*. IEEE Transactions on VLSI Systems, vol. 3, no. 3, pages 459–464, September 1995.

[van der Wolf 04] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer & Gerben Essink. *Design and programming of embedded multiprocessors: an interface-centric approach*. In Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04), pages 206–217, New York, USA, 2004. ACM.

[van Eijndhoven 99] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel & H. P. E. Vranken. *TriMedia CPU64 Architecture*. In Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers and Processors (ICCD'99), 1999.

[Vannel 07] Fabien Vannel. *Bio-inspired cellular machines: towards a new electronic paper architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, 2007.

[Von Neumann 66] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[Wiangtong 02] Theerayod Wiangtong, Peter Y.K. Cheung & Wayne Luk. *Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign*. Design Automation for Embedded Systems, vol. 6, no. 4, pages 425–449, July 2002.

[Wiangtong 04] Theerayod Wiangtong. *Hardware/Software Partitioning And Scheduling For Reconfigurable Systems*. PhD thesis, Imperial College London, February 2004.

[Wiklund 03] Daniel Wiklund & Dake Liu. *SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems*. In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03), page 78.1, Washington, USA, 2003. IEEE Computer Society.

[Xie 01] Y. Xie, W. Wolf & H. Lekatsas. *Code Compression for VLIW Processors*. In Proceedings of the 34th International Symposium on Microarchitecture (MICRO 01), pages 66–75, 2001.

[Xil 07] Xilinx. *Xilinx Power Estimator User Guide*, 2007. `http://www.xilinx.com/products/design_resources/power_central/ug440.pdf`.

[Ye 03] Terry Tao Ye, Luca Benini & Giovanni de Micheli. *Packetized On-Chip Interconnect Communication Analysis for MPSoC*. In Proceedings of the conference on Design, automation and test in Europe (DATE'03), vol. 1, pages 344–349, Los Alamitos, CA, USA, March 2003. IEEE Computer Society.

Appendix A

Processor implementation appendix

A.1 Bus interface VHDL entity

```
entity bus_io_interface is
generic
(
        -- The base address the unit has
        base_address : type_base_address := 1;

        -- The number of page addresses ( adressable registers )
        -- accessible in the functional unit
        n_registers : positive := 1
);
port
(
        ---------------------------------
        -- Transport bus interfacing --
        ---------------------------------
        io_bus : inout data_word_array;
        src : in unit_address_word_array;
        dest : in unit_address_word_array;

        ----------------------
        -- FU interfacing --
        ----------------------
        -- Asserted when registers are set as destination and
        -- should be written
        we_regs :       out std_logic_vector(n_registers - 1 downto 0);
        -- Asserted when registers are set as source
        re_regs :       out std_logic_vector(n_registers - 1 downto 0);

        -- The registers next input
        reg_inputs : out matrix_array(0 to n_registers - 1);

        -- The registers output of the FU
        reg_outputs : in matrix_array(0 to n_registers - 1)
);
end;
```

Listing A.1: VHDL entity of the bus interface.

A.2 Functional units memory map

A.2.1 Arithmetic and logic unit

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	No	A	Operand A
1	RW	Yes	B	Operand B
2	R	No	ADD	$A + B$
3	R	No	SUB	$A - B$
4	R	No	OR	$A \vee B$
5	R	No	AND	$A \wedge B$
6	R	No	NOT	$\neg B$
7	R	No	XOR	$A \oplus B$
8	R	No	MULT_{low}	$(A \cdot B)_{[31...0]}$
9	R	No	MULT_{high}	$(A \cdot B)_{[63...32]}$

Table A.1: ALU functional unit memory map.

A.2.2 Shift parallel unit

A generic parameter can configure the parallel shift unit to include or not the rotation operation.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	A	Operand A
1	RW	No	N	Number of bits for the operation
2	R	No	Shift right	$A \gg N$
3	R	No	Shift right arithmetic	$(A \gg N)_{arith}$
4	R	No	Shift left	$A \ll N$
5	R	No	Rotate left	$A \curvearrowleft B$
6	R	No	Rotate right	$A \curvearrowright A$

Table A.2: Parallel shifter memory map.

A.2.3 Concatenation

This unit is automatically used by the assembler to generate long immediates, as described in section 3.8.1.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	W	No	A	Operand A, width = size of short immediate value
1	W	No	B	Operand B, width = size of short immediate value
0	R	No	OUT	$Out = (A \ll 16) \vee B$
1	R	No	ZERO	Always zero

Table A.3: Concatenation unit memory map.

A.2.4 Comparison / Condition unit

This unit is mostly used to generate conditions for jumps, as described in section 3.8.2.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	OP_A	Operand A
1	RW	No	OP_B	Operand B
2	RW	No	OPCODE	Selects the operation performed
3	R	No	RESULT	Depending on OPCODE: $A < 0$, $A > 0$, $A = 0$, $A \neq 0$, $A > B$, $A \geq B$, $A < B$, $A \leq B$, $A = B$, $A \neq B$

Table A.4: Condition functional unit memory map.

A.2.5 GPIO

This unit can be used to provide general-purpose I/O pins to the processor to interface to the outside world.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	GPIO_OUT	Output word on GPIO pins. The number of GPIO pins for output is generic.
1	R	No	GPIO_IN	Value read on GPIO pins. The number of GPIO pins for input is generic.

Table A.5: GPIO functional unit memory map.

A.2.6 MERCURY unit

This unit is used to interface to the CONFETTI subsystem *via* the standard MERCURY interface, as described in section 8.4.1.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	Yes	STATUS	The status register of the unit[a].
1	RW	No	DMA_{start}	Start address of DMA for MERCURY to memory write.
2	RW	No	DMA_{end}	Max address allowed for DMA write.
3	W	Yes	RST	Any value written resets the new data flag.
3	R	No	SIZE	The size of the last received packet.
4	W	Yes	SEND	Pass the written data to the MERCURY sender.
4	R	No	HEADER	Header of the last received packet.

[a]
- `ACCEPT_DMA_MASK` (bit 0, lsb): 1 if the unit should accept new incoming data, if set to 0 the unit receiver keeps its `RDY` line deasserted;
- `ALLOW_RECONFIG` (bit 1): 1 if the unit is allowed to accept new code and reboot, if set to 0 the arrival of a packet with a new code will be ignored;
- `WRITE_IN_PROGRESS` (bit 2): 1 if MERCURY is currently receiving data;
- `NEW_DATA_FLAG` (bit 3): 1 if MERCURY receiver has finished writing a whole packet to memory;
- `CAN_SEND_DATA` (bit 4): 1 if MERCURY sender is ready to handle a new packet of data;
- `RECEIVER_EMPTY` (bit 5): 1 if there are data in the receiver waiting to be copied with DMA.

Table A.6: Mercury FU memory map.

A.2.7 Display interface

This unit interfaces with the display area on top of the cell in the CONFETTI platform, as described in section 8.4.2.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	R	No	X	X-coordinate of this CPU in the CONFETTI grid.
1	R	No	Y	Y-coordinate of this CPU in the CONFETTI grid.
0	W	No	ADDR	Address of the pixel to modify.
1	W	Yes	COLOR	Sets the color of the pixel.
2	R	No	TOUCH	# of seconds the touch sensor has been pressed continuously.
3	R	No	GRAPHIC	Last pixel value written to the graphic memory.
[4...7]	RW	No	GR[0...3]	General purpose register [0...3].

Table A.7: Display FU memory map.

A.2.8 Timer

This unit can be used to accurately measure time with a µs resolution. The unit uses a generic definition of the processor speed, provided in the entity of the unit, to accomodate different clock periods. More precisely, the counter is updated every $(\frac{clk\_frequency}{10^6} - 1)$.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	R	No	US_COUNTER	Number of µs since last processor or timer reset.
0	W	Yes	RST	Resets the timer counter.

Table A.8: Timer FU memory map.

A.2.9 SRAM interface

This functional unit can be used to connect asynchronous SRAM memory to ULYSSE, as described in section 3.7.2.

Address	*RW mode*	*Trigger*	*Name – Function*	*Description*
0	RW	No	SP1	Stack pointer number 1
1	RW	No	SP2	Stack pointer number 2
2	RW	No	DATA	Memory data register
3	W	Yes	Load absolute	Data ← Memory(Address)
4	W	Yes	Store absolute	Memory(Address) ← Data
5	W	Yes	Load with offset 1	Data ← Mem(Address + SP1)
6	W	Yes	Load with offset 2	Data ← Mem(Address + SP2)
7	W	Yes	Store with offset 1	Mem(Address + SP1) ← Data
8	W	Yes	Store with offset 2	Mem(Address + SP2) ← Data

Table A.9: SRAM FU Interface memory map.

A.2.10 Assertion unit

This unit is used in the context of asserted simulation (see section 5.2).

Address	*RW mode*	*Trigger*	*Name – Function*	*Description*
0	W	No	A	Operand A
1	W	No	B	Operand B
2	W	No	LINE	Line of code currently asserted (to be displayed in MODELSIM^{TM} in case of error).
3	W	Yes	STOP	Stop simulation in MODELSIM^{TM} at current line.
4	W	Yes	DUMP$_{start}$	Start dumping memory in MODELSIM^{TM} with address DUMP$_{start}$.
5	W	Yes	DUMP$_{address}$	Display memory content of address DUMP$_{address}$ in MODELSIM^{TM}.
6	W	Yes	DUMP$_{stop}$	Stop dumping memory in MODELSIM^{TM}.

Table A.10: Assertion FU memory map.

A.2.11 Divider unit

This unit is articulated around an IP module published by Xilinx and accessible with its *CoreGen* tool. As this divider is pipelined, maximum throughput is achieved when data are input every cycle and divisions made in a row, especially since initial latency in the chosen configuration[1] is 41 cycles. As the module does not provide a signal to indicate valid results, we had to implement a state machine that counts the number of cycle corresponding to latency to know when the operator completes, asserting the busy bit of the unit that time.

Address	*RW mode*	*Trigger*	*Name*	*Description*
0	RW	No	OP_A	Dividend
1	RW	Yes	OP_B	Divisor
2	R	No	QUOTIENT	$Dividend/Divisor$, with "/" integer division
3	R	No	REMAINDER	$Dividend - Quotient \cdot Divisor$

Table A.11: Divider unit memory map.

[1] Dividend, divisor, remainder and quotient all on 32 bits

A.3 Functional unit code samples

A.3.1 Concatenation unit

```
--
-- EPFL
-- Ulysse processor - Concatenation unit for 16 bits immediate values
-- Pierre-Andre Mudry, 2008
--

library ieee, ulysse;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;
use ulysse.ulysse_utils.all;

architecture str of concat is
  signal reg_0_in, reg_0_out, reg_1_in, reg_1_out : type_immediate;
  signal big_reg_0_in, big_reg_1_in : data_word;

  subtype data_range is integer range immediate_size - 1 downto 0;
  signal reg_out : data_word;
  signal we_regs : std_logic_vector(1 downto 0);
  signal we_0, we_1 : std_logic;
begin

-- Asserted when writing to page 0
we_0 <= we_regs(0);
-- Asserted when writing to page 1
we_1 <= we_regs(1);

-- Size mapping
reg_0_in <= big_reg_0_in(data_range);
reg_1_in <= big_reg_1_in(data_range);

-- This FU is never busy, latency 1
busy <= '0';

-- Register to store the low immediate value
i_reg_0 : entity ulysse.reg generic map (register_width => immediate_size)
          port map (clk => clk, we => we_0, rst => rst, input => reg_0_in,
          output => reg_0_out);

-- Register to store the high immediate value
i_reg_1 : entity ulysse.reg generic map (register_width => immediate_size)
          port map (clk => clk, we => we_1, rst => rst, input => reg_1_in,
          output => reg_1_out);

-- Instantiate the bus interface
i_bus_in : entity ulysse.bus_io_interface generic map
           (base_address => base_address, n_registers => 2) port map
           (io_bus => io_bus,
                      src => src,
                      dest => dest,
                      we_regs => we_regs,
                      re_regs => open,
                      reg_inputs(0) => big_reg_0_in,
                      reg_inputs(1) => big_reg_1_in,
```

```
55                              reg_outputs(0) => reg_out,
56                              reg_outputs(1) => zero_dw -- Always zero
57              );
58
59  -- Concatenate the two immediate values
60  process(reg_1_out, reg_0_out)
61  begin
62    reg_out <= reg_1_out & reg_0_out;
63  end process;
64
65  end str;
```

Listing A.2: Concatenation unit implementation.

A.3.2 The fetch unit

```
--
-- EPFL, GRIJ
-- Ulysse processor 2.1
-- Pierre-Andre Mudry, March 2008
--
-- The fetch unit grabs new instructions from the code memory
-- and, after a simple decoding, sends them to the addresses and data busses.
-- The halt inputs can be asserted to stop the processor operation.
--

library ieee, ulysse;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ulysse.ulysse_utils.all;

-- pragma translate_off
use ulysse.image_pkg.all;
use std.textio.all;
-- pragma translate_on

architecture str of fetch_unit32 is
  signal pc_in, reg_pc : address_word;
  signal ir_in, reg_ir : instruction_word;

  -- This signal always contain the currently executed instruction
  signal active_ir : instruction_word;

  signal we_pc, we_ir : std_logic;
  signal pc_external_offset, pc_external_immediate : std_logic;

  signal src_int, dest_int : unit_address_word;
  signal src_page, dest_page : integer range 15 downto 0;
  signal src_base, dest_base : integer range 15 downto 0;

  signal just_reset : std_logic;
  signal reset_cpu : std_logic;
  signal io_direction_internal : std_logic;

  -- Conditional jumps condition
  signal we_condition : std_logic;
  signal condition_in, condition_in1 : std_logic;
  signal pc_jump_with_condition : std_logic;
  signal reg_condition : std_logic_vector(0 downto 0);

  signal is_immediate : std_logic;

  -- RET register used to store the return address of a function
  signal ret_in, reg_ret : address_word;
  signal we_ret : std_logic;

  -- Fetch unit state management
  type fetch_state_type is (fetch1, fetch2, fetch3, execute, stall, debug_state);
  signal fetch_state, next_fetch_state : fetch_state_type;

  -- Enables the write of the IR when the new instruction has been fetched
  signal we_ir_sm : std_logic;

  -- Enables the write of the PC only when the new instruction has been fetched
  signal we_pc_sm : std_logic;

  -- Indicates that the current instruction accesses a busy FU
  signal axxs_busy_fu : std_logic;

  ----------------------
```

```
67    -- Debugging related
68    ----------------------
69    -- pragma translate_off
70    -- Stores the instruction strings for debugging purposes
71    type array_instr is array (integer range <>) of string(1 to 60);
72    signal currentInstr : string(1 to 60);
73
74    -- Reads the file containing the instructions in a textual form so they
75    -- can be displayed directly in the simulator
76    procedure readIstrFile(instr : inout array_instr) is
77      file data_file : text open read_mode is debug_istr_fileName;
78      variable l : line;
79      variable current_address : natural := 0;
80      variable s : string(1 to 60);
81    begin
82      While Not Endfile(data_file) Loop
83          s := (others => ' ');
84          Readline(data_file, l);
85          Read(l, s(l'range)); -- Reads the string from the line
86          instr(current_address) := s;
87          current_address := current_address + 1;
88      End Loop;
89    end procedure readIstrFile;
90    -- pragma translate_on
91
92  begin
93
94  reset_cpu <= restart_cpu or rst;
95
96  ----------------------------------------
97  -- Registers present in the fetch unit
98  ----------------------------------------
99  i_pc : entity ulysse.reg generic map (register_width => (memory_height))
100             port map (clk => clk, we => we_pc, rst => reset_cpu, input => pc_in, output => reg_pc);
101
102 i_ir : entity ulysse.reg generic map (register_width => instruction_width*instruction_slots)
103             port map (clk => clk, rst => reset_cpu, we => we_ir, input => ir_in, output => reg_ir);
104
105 i_condition : entity ulysse.regbit
106             port map (clk => clk, rst => reset_cpu, we => we_condition, input => condition_in, output => reg_condition(0));
107
108 i_jrst : entity ulysse.regbit
109             port map (clk => clk, rst => '0', we => '1', input => rst, output => just_reset);
110
111 i_ret : entity ulysse.reg generic map(register_width => memory_height)
112             port map (clk => clk, rst => reset_cpu, we => we_ret, input => ret_in, output => reg_ret);
113
114 assert(current_n_fu = 4 or current_n_fu = 8 or current_n_fu = 16 or current_n_fu = 32 or current_n_fu = 64)
115   report string'("Number of FU specified not a power of 2 or not < 128") severity failure;
116
117
118 -- Halt and write signals must be able to reflect the busy conditions coming from the
119 -- functional units
120 we_ir <= (not halt) and we_ir_sm;--and not(axxs_busy_fu);
121
122 -- The IR is the content of the memory location pointed by the PC
123 ir_in <= mem_q;
124
125 -----
126 -- BUSY FU Management
127 -- The idea here is when an operation takes more than one cycle it asserts
128 -- it's busy flag and, while no other operation tries to read or write to
129 -- the busy FU, the processor behaves normally. However, if an operation
130 -- needs to read or write the busy FU, the processor is halted until
131 -- the FU is no more busy in order to resolve the dependency automatically.
132 -----
133 -- Is the currently active instruction containing an immediate value ?
134 process(active_ir)
```

```
135 begin
136 is_immediate <= '0';
137 if ( active_ir (is_immediate_field) = '1') then
138   is_immediate <= '1';
139 end if;
140 end process;
141
142 -- SRC and DST page and base extraction
143 process(src_int, dest_int)
144 begin
145   src_page <= extract_page(src_int);
146   dest_page <= extract_page(dest_int);
147
148   src_base <= to_integer(ieee.numeric_std.unsigned(src_int(7 downto 4)));
149   dest_base <= to_integer(ieee.numeric_std.unsigned(dest_int(7 downto 4)));
150 end process;
151
152 -- IO Process for transport bus 1
153 process(active_ir, reg_pc, is_immediate, io_bus, halt,
154           reg_ret, fetch_state, reg_ir, src_base, src_page,
155           dest_base, dest_page)
156 begin
157 io_bus(0) <= z_dw;
158
159 pc_external_offset <= '0';
160 pc_external_immediate <= '0';
161 pc_jump_with_condition <= '0';
162 condition_in1 <= '0';
163
164 -- RET
165 we_ret <= '0';
166 ret_in <= (others => '0');
167
168 -- Default is to read the io_bus
169 io_direction_internal <= '1';
170
171 -- If the fetch unit is used as a source
172 if (src_base = base_address and fetch_state = execute) then
173   io_direction_internal <= '0';
174
175   -- Output to bus the immediate value contained in the instruction
176   if (is_immediate = '1') then
177     io_bus(0) <= sxt( active_ir (immediate_field), data_width);
178   else
179     -- Output to the bus the PC or the IR
180     case(src_page) is
181       when 1 => io_bus(0) <= pad_with_zeroes(reg_pc, data_width);
182       when 2 => io_bus(0) <= pad_with_zeroes(reg_ir, data_width);
183       when 3 => io_bus(0) <= pad_with_zeroes(reg_ret, data_width);
184       --when 4 => io_bus(0) <= ext(processor_id, data_width);
185       when others => null;
186     end case;
187   end if;
188 end if;
189
190 -- PC modifications from the outside ( relative and absolute jumps)
191 -- Aka the fetch unit is used as a destination
192 if (dest_base = base_address and halt = '0') then
193   -- Input from bus
194   case(dest_page) is
195     when 1 =>
196       pc_external_offset <= '1';
197     when 2 =>
198       pc_external_immediate <= '1';
199     when 3 =>
200       ret_in <= io_bus(0)(memory_height - 1 downto 0);
201       we_ret <= '1';
202     when 4 =>
```

```
203         pc_jump_with_condition <= '1';
204       when 5 =>
205         condition_in1 <= '1';
206       when others => null;
207     end case;
208 end if;
209
210 end process;
211
212 -- High-impedance selection
213 io_direction  <= io_direction_internal;
214
215 -- Condition register update & arbitration
216 process(condition_in1, io_bus)
217 begin
218 we_condition <= '0';
219 condition_in <= '0';
220
221 if (condition_in1 = '1') then
222     we_condition <= '1';
223
224     -- The data can come from an immediate, which is
225     -- stored in the instruction (hence io_bus(0)) or from
226     -- an FU --> io_bus_in
227     condition_in <= io_bus(0)(0);
228 end if;
229
230 end process;
231
232 ------------------------------
233 -- PC Update & arbitration
234 ------------------------------
235 process(just_reset, rst, reg_pc, io_bus, pc_external_immediate,
236           pc_jump_with_condition, reg_condition, fetch_state, restart_cpu)
237   variable tmp : data_word;
238 begin
239   we_pc <= '0';
240
241   if (fetch_state = execute or just_reset = '1' or restart_cpu = '1') then
242     we_pc <= '1';
243   end if;
244
245   -- Automatic PC update (normal case operation)
246   pc_in <= reg_pc + 1;
247
248   -- Immediate value for PC replacement (non relative jump)
249   if (pc_external_immediate = '1') then
250     pc_in <= io_bus(0)(memory_height - 1 downto 0);
251   end if;
252
253   -- Conditional jumps
254   if (pc_jump_with_condition = '1' and reg_condition(0) = '1') then
255     pc_in <= io_bus(0)(memory_height - 1 downto 0);
256   end if;
257
258   -- PC default value
259   if (just_reset = '1' or rst = '1' or restart_cpu = '1') then
260     pc_in <= (others => '0');
261   end if;
262 end process;
263
264 ------------------------------
265 -- Assertions using PSL
266 ------------------------------
267 -- Never access a FU with address > allowed number of fus
268 -- psl assert always (rst = '0' -> (src(0)(base_range) <= current_n_fu))
269 --     report "Bad src address range";
270 -- psl assert always (rst = '0' -> (dest(0)(base_range) <= current_n_fu))
```

```
271 --      report "Bad dest address range";
272 process(src_int, dest_int, fetch_state)
273 begin
274 if (fetch_state = execute) then
275   src(0) <= pad_with_zeroes(src_int(7 downto 0), 15);
276   dest(0) <= pad_with_zeroes(dest_int(7 downto 0), 15);
277 else
278   src(0) <= (others => '0');
279   dest(0) <= (others => '0');
280 end if;
281 end process;
282
283 -- Code memory interface
284 process(reg_pc)
285 begin
286   mem_address <= reg_pc;
287 end process;
288
289 process(fetch_state, ir_in, reg_ir)
290 begin
291 if (fetch_state = execute) then
292   active_ir <= ir_in;
293 else
294   active_ir <= reg_ir;
295 end if;
296 end process;
297
298 ------------------------------
299 -- Fetch unit state machine
300 ------------------------------
301 fetch_sm :
302 process(fetch_state, busy_flags, is_immediate, active_ir, dest_base, src_base, halt)
303 begin
304 mem_req <= '0';
305 we_ir_sm <= '0';
306 we_pc_sm <= '0';
307 axxs_busy_fu <= '0';
308
309 src_int <= (others => '0');
310 dest_int <= (others => '0');
311
312   case fetch_state is
313     when fetch1 =>
314       debug_out <= x"1";
315
316       if (halt = '1') then
317         next_fetch_state <= debug_state;
318       else
319         we_pc_sm <= '1';
320         next_fetch_state <= fetch2;
321         mem_req <= '1';
322       end if;
323
324     when fetch2 =>
325       debug_out <= x"2";
326       mem_req <= '1';
327       next_fetch_state <= fetch3;
328
329     when fetch3 =>
330       debug_out <= x"3";
331       mem_req <= '1';
332       next_fetch_state <= execute;
333
334     when execute =>
335       debug_out <= x"4";
336       we_ir_sm <= '1';
337
338       -- Normal case
```

```
339         next_fetch_state <= fetch1;
340         mem_req <= '1'; -- Indicate memory that next state should be loading
341
342         -- Internal source and destination busses assignment
343         dest_int <= active_ir(destination_field);
344         if (is_immediate = '0') then
345           src_int <= active_ir(source_field);
346         end if;
347
348         -- If the destination we want is busy,
349         -- then stall until free again
350         if (busy_flags(dest_base) = '1') then
351           next_fetch_state <= stall;
352           axxs_busy_fu <= '1';
353           mem_req <= '0';
354         elsif (is_immediate = '0' and busy_flags(src_base) = '1') then
355           -- If the source we want is busy (and not an immediate),
356           -- then stall until free again
357           axxs_busy_fu <= '1';
358           next_fetch_state <= stall;
359           mem_req <= '0';
360         end if;
361
362       when stall =>
363         debug_out <= x"5";
364         dest_int <= active_ir(destination_field);
365
366         if (is_immediate = '0') then
367           src_int <= active_ir(source_field);
368         end if;
369
370         -- If the destination we want is busy,
371         -- then stall until free again
372         if (busy_flags(dest_base) = '1') then
373           next_fetch_state <= stall;
374           axxs_busy_fu <= '1';
375         elsif (is_immediate = '0' and busy_flags(src_base) = '1') then
376          -- If the source we want is busy (and not an immediate),
377          -- then stall until free again
378           next_fetch_state <= stall;
379           axxs_busy_fu <= '1';
380         else
381           next_fetch_state <= fetch1;
382           mem_req <= '1'; -- Indicate memory that next state should be loading
383         end if;
384
385       when debug_state =>
386         debug_out <= x"6";
387         next_fetch_state <= debug_state;
388
389         if (halt = '0') then
390           next_fetch_state <= fetch1;
391           mem_req <= '1'; -- Indicate memory that next state should be loading
392         end if;
393
394     end case;
395   end process;
396
397   -- Fetch unit State machine updater
398   fetch_state_updater:
399   process(clk, rst)
400   begin
401     if (rst = '1') then
402       fetch_state <= fetch1;
403     elsif (rising_edge(clk)) then
404       fetch_state <= next_fetch_state;
405     end if;
406   end process;
```

```
407
408 ------------------------------
409 -- The rest of this file is used only during debugging to help
410 -- the programmer.
411 ------------------------------
412 -- pragma translate_off
413 process(reg_pc, fetch_state, pc_in)
414   variable instructions : array_instr(0 to 262143);
415   variable initDone : boolean := false;
416 begin
417   -- At first call, load the file containing the textual
418   -- representation of the instructions and put it in the
419   -- instructions array
420   if(not initDone) then
421     readIstrFile(instructions);
422     currentInstr <= instructions(0);
423     initDone := true;
424   else
425     if(fetch_state = execute) then
426       currentInstr <= instructions(to_integer(ieee.numeric_std.unsigned(pc_in)));
427     else
428       currentInstr <= instructions(to_integer(ieee.numeric_std.unsigned(reg_pc)));
429     end if;
430   end if;
431 end process;
432
433 -- Dumps various debugging information such as
434 -- number of cycles, current instruction ... in the simulator
435 process(clk)
436   variable d : line;
437   variable clk_cycle : integer := 0;
438   variable clk_cycle_since_rst : integer := 0;
439   variable cycle_count : integer := 0;
440 begin
441 if(rising_edge(clk)) then
442   if(rst = '0') then
443     clk_cycle := clk_cycle + 1;
444     clk_cycle_since_rst := clk_cycle_since_rst + 1;
445
446     if(dump_instr and next_fetch_state = execute) then
447       cycle_count := cycle_count + 1;
448       write(d, string'("* Clk since begin " & integer'image(clk_cycle) & " / Total clk " & integer'image(
              clk_cycle_since_rst) & " - cycle " & integer'image(cycle_count)));
449       writeline(output, d);
450       write(d, string'("    * Execute " & currentInstr));
451       writeline(output, d);
452     end if;
453
454     if(dump_instr = false and next_fetch_state = execute) then
455       if(clk_cycle_since_rst mod 1000 = 0) then
456         write(d, string'("+"));
457         writeline(output, d);
458       end if;
459     end if;
460   else
461     clk_cycle_since_rst := 0;
462     cycle_count := 0;
463   end if;
464 end if;
465 end process;
466 -- pragma translate_on
467 end str;
```

Listing A.3: Fetch unit code.

A.3.3 The memory unit

```
library ieee, ulysse;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ulysse.ulysse_utils.all;

-- pragma translate_off
use ulysse.image_pkg.all;
use std.textio.all;
-- pragma translate_on

entity harvard_memory_unit is
    generic
    (
      base_address : type_base_address := 12
    );
    port
    (
      clk : in std_logic;
      clk4x : in std_logic;
      rst : in std_logic;
      is_idle : out std_logic;

      -- Ulysse standard access
      busy : out std_logic;
      io_bus : inout data_word_array;
      src : in unit_address_word_array;
      dest : in unit_address_word_array;

      -- Connexion to fetch unit
      fetch_mem_address : in address_word;
      fetch_mem_data : out instruction_word;
      fetch_mem_req : in std_logic;

      -- External control of memory
      external_req : in std_logic;
      external_address : in address_word;
      external_data : in instruction_word;
      external_rw : in std_logic; -- Direction of transfer

      -- To the SRAMs IO Ports
      a : out address_word;
      d : inout std_logic_vector(half_instruction_width - 1 downto 0);
      nCS0 : out std_logic;
      nCS1 : out std_logic;
      nWE : out std_logic;
      nOE : out std_logic;
      nLB : out std_logic;
      nUB : out std_logic
    );
end;

architecture str of harvard_memory_unit is
  signal enable, wr : std_logic;

  signal data_towrite, data_read : data_word;
  signal address : address_word;

  type state_type is (idle, fetch1, fetch2, fetch3, load1, load2, load3, store_operation, prestore_operation);
  signal state, next_state : state_type;

  signal we_sp1, we_sp2, we_data : std_logic;
  signal re_sp1, re_sp2, re_data : std_logic;
  signal sp1_in, reg_sp1 : address_word;
  signal sp2_in, reg_sp2 : address_word;
```

```
67    signal data_in, reg_data : data_word;
68
69    signal big_sp1_in, big_sp2_in, big_reg_sp1, big_reg_sp2, data_in_fu : data_word;
70
71    signal we_data_ext, we_data_int : std_logic;
72
73    -- Signals generated by the functional unit
74    signal fu_mem_address : address_word;
75    signal fu_wr : std_logic;
76
77    signal we_fu_mem_address : std_logic;
78    signal fu_mem_address_reg : address_word;
79
80    signal nil_sig : std_logic;
81    signal nul_word, nul_2 : data_word;
82    type type_nul_array is array (0 to 8) of data_word;
83    signal nul_array_in : type_nul_array;
84    signal nul_array_out : type_nul_array;
85
86    -- Memory operation signals
87    signal load_abs_addr, write_abs_addr, load_off_sp1, load_off_sp2, write_off_sp1, write_off_sp2 : std_logic;
88    signal load_op : std_logic; -- Asserted when a memory loading operation (from FU) is taking place
89    signal store_op : std_logic; -- Asserted when a memory store operataion (from FU) is taking place
90    signal adder_address : address_word; -- Contains the computed address for the current operation
91    signal to_add : address_word; -- Contains the signal that should be added to compute current address accessed
92
93    -- We have to update the data reg when this signal is valid
94    signal we_read_data_valid, read_data_valid, read_data_valid_reg : std_logic;
95
96    signal ext_store : std_logic;
97
98    signal we_address : std_logic;
99    signal adder_address_reg : address_word;
100   --
101   signal external_req_reg : std_logic;
102 -- signal we_already_accessed , already_accessed , already_accessed_reg : std_logic ;
103
104   signal noe_gen, noe_early : std_logic;
105 begin
106
107 nul_word <= (others => 'Z');
108 we_data <= we_data_ext or we_data_int;
109
110 -- Stack pointers
111 i_sp1:
112             entity ulysse.reg generic map (register_width => memory_height)
113             port map (clk => clk, rst => rst, we => we_sp1, input => sp1_in, output => reg_sp1);
114
115 i_sp2:
116             entity ulysse.reg generic map (register_width => memory_height)
117             port map (clk => clk, rst => rst, we => we_sp2, input => sp2_in, output => reg_sp2);
118
119 i_fu_mem_address:
120             entity ulysse.reg generic map (register_width => memory_height)
121             port map (clk => clk, rst => rst, we => we_fu_mem_address, input => fu_mem_address, output =>
                    fu_mem_address_reg);
122
123 i_data:
124             entity ulysse.reg generic map (register_width => data_width)
125             port map (clk => clk, rst => rst, we => we_data, input => data_in, output => reg_data);
126
127 i_data_valid:
128             entity ulysse.regbit
129             port map (clk => clk, rst => rst, we => we_read_data_valid, input => read_data_valid, output =>
                    read_data_valid_reg);
130
131 i_bus_in :
132             entity ulysse.bus_io_interface generic map
```

```
133             (base_address => base_address, n_registers => 9) port map
134             (io_bus => io_bus,
135              src  => src,
136              dest  => dest,
137              we_regs(0) => we_sp1, -- sp1 = io_bus
138              we_regs(1) => we_sp2, -- sp2 = io_bus
139              we_regs(2) => we_data_ext,-- data = io_bus
140              we_regs(3) => load_abs_addr, -- data = mem(abs)
141              we_regs(4) => write_abs_addr, -- mem(abs) = data
142              we_regs(5) => load_off_sp1,   -- data = mem(SP1 + offset1)
143              we_regs(6) => load_off_sp2,   -- data = mem(SP2 + offset2)
144              we_regs(7) => write_off_sp1,  -- mem(SP1 + offset1) = data
145              we_regs(8) => write_off_sp2,  -- mem(SP2 + offset2) = data
146
147              re_regs(0)  => re_sp1, -- io_bus = sp1
148              re_regs(1)  => re_sp2, -- io_bus = sp2
149              re_regs(2)  => re_data, -- io_bus = data
150              re_regs(3)  => nil_sig,
151              re_regs(4)  => nil_sig,
152              re_regs(5)  => nil_sig,
153              re_regs(6)  => nil_sig,
154              re_regs(7)  => nil_sig,
155              re_regs(8)  => nil_sig,
156
157              reg_inputs(0)  => big_sp1_in,
158              reg_inputs(1)  => big_sp2_in,
159              reg_inputs(2)  => data_in_fu,
160              reg_inputs(3)  => nul_array_in(3),
161              reg_inputs(4)  => nul_array_in(4),
162              reg_inputs(5)  => nul_array_in(5),
163              reg_inputs(6)  => nul_array_in(6),
164              reg_inputs(7)  => nul_array_in(7),
165              reg_inputs(8)  => nul_array_in(8),
166
167              reg_outputs(0) => big_reg_sp1,
168              reg_outputs(1) => big_reg_sp2,
169              reg_outputs(2) => reg_data,
170              reg_outputs(3) => nul_array_out(3),
171              reg_outputs(4) => nul_array_out(4),
172              reg_outputs(5) => nul_array_out(5),
173              reg_outputs(6) => nul_array_out(6),
174              reg_outputs(7) => nul_array_out(7),
175              reg_outputs(8) => nul_array_out(8)
176             );
177
178 sp1_in <= big_sp1_in(memory_height - 1 downto 0);
179 sp2_in <= big_sp2_in(memory_height - 1 downto 0);
180 big_reg_sp1 <= ext(reg_sp1, data_width);
181 big_reg_sp2 <= ext(reg_sp2, data_width);
182
183 -- Asserted when a FU loading operation is taking place
184 load_op  <= load_off_sp1 or load_off_sp2 or load_abs_addr;
185 store_op <= write_abs_addr or write_off_sp1 or write_off_sp2;
186
187 sram_controller :
188   entity ulysse.sram_controller(str) port map
189     (
190       clk  => clk,
191       clk4x => clk4x,
192       rst  => rst,
193       -- Ctrl signals
194       enable => enable,
195       address => address,
196       data_towrite => data_towrite,
197       data_read => data_read,
198       wr => wr,
199       -- SRAM IO pins
200       nCS0_reg => nCS0,
```

```
201         nCS1_reg => nCS1,
202         a  => a,
203         d => d,
204         nWE => nWE,
205         nOE => nOE_gen,
206         nLB => nLB,
207         nUB => nUB
208       );
209
210 datasrc_chooser:
211 process(external_req, external_data, we_data_int, data_read, data_in_fu)
212 begin
213 if (external_req = '1') then
214   data_in <= external_data;
215 elsif (we_data_int = '1') then
216   data_in <= data_read;
217 else
218   data_in <= data_in_fu;
219 end if;
220 end process;
221
222 ------------------------------
223 -- Assertions using PSL
224 ------------------------------
225
226 -- Make sure that the address won't change when fetching a new data
227 ---- psl assert always ((rst = '0' and fetch_mem_req = '1') -> next(prev(fetch_mem_address) = fetch_mem_address))@(
        rising_edge(clk))
228 ----   report "Address not stable";
229
230 nOE <= noe_gen or noe_early;
231
232 process(state, fetch_mem_req, fetch_mem_address,
233           data_read, read_data_valid_reg,
234           load_op, store_op, external_req,
235           reg_data, adder_address_reg)
236 begin
237 -- Memory ctrl
238 enable <= '0';
239 address <= (others => '0');
240 wr <= '0';
241 data_towrite <= reg_data;
242
243 -- Data register input
244 we_data_int <= '0';
245
246 -- Ctrl for the various FU operations
247 fu_mem_address <= (others => '0');
248 fu_wr <= '0';
249 we_fu_mem_address <= '0';
250
251 we_address <= '0';
252
253 busy <= '0';
254
255 we_read_data_valid <= '0';
256 read_data_valid <= '0';
257 fetch_mem_data <= data_read;
258
259 ext_store <= '0';
260 is_idle <= '0';
261
262 noe_early <= '0';
263
264   case state is
265     when idle =>
266       next_state <= idle;
267       is_idle <= '1';
```

```
268
269         -- Register the data read if previous operation was a read
270         if (read_data_valid_reg = '1') then
271           assert not(is_x(data_read)) report
272             string'("Data read not valid. Memory not initialized before read ?")
273             severity failure ;
274           we_read_data_valid <= '1';
275           read_data_valid <= '0';
276           we_data_int <= '1'; -- Data is now valid
277           is_idle <= '0';
278         end if;
279
280         -- If memory is controlled from the outside of the CPU
281         -- let's be controlled from the outside
282         if (external_req = '1') then
283           -- Write current data
284           we_data_int <= '1';
285           next_state <= prestore_operation;
286           we_address <= '1';
287           ext_store <= '1';
288         else
289           -- Fetch unit requests access to the RAM
290           if (fetch_mem_req = '1') then
291             next_state <= fetch1;
292           end if;
293
294           -- Excecute phase requests a load
295           if (load_op = '1') then
296             next_state <= load1;
297             we_address <= '1';
298             busy <= '1';
299           end if;
300
301           -- Execute phase request a write
302           if (store_op = '1') then
303             next_state <= prestore_operation;
304             we_address <= '1';
305             busy <= '1';
306           end if;
307         end if;
308
309       when fetch1 =>
310         -- Register the data read if previous operation was a read
311         if (read_data_valid_reg = '1') then
312           assert not(is_x(data_read)) report
313             string'("Data read not valid. Memory not initialized before read ?")
314             severity failure ;
315           we_read_data_valid <= '1';
316           read_data_valid <= '0';
317           we_data_int <= '1'; -- Data is now valid
318         end if;
319
320         noe_early <= '0';
321         wr <= '0';
322         enable <= '1';
323         address <= fetch_mem_address;
324         next_state <= fetch2;
325
326       -- When the fetch unit has the control
327       when fetch2 =>
328         noe_early <= '0';
329         address <= fetch_mem_address;
330         next_state <= fetch3;
331
332       -- When the fetch unit has the control
333       when fetch3 =>
334         noe_early <= '0';
335         address <= fetch_mem_address;
```

```
336         next_state <= idle;
337
338       when prestore_operation =>
339         noe_early <= '1';
340         enable <= '1';
341         wr <= '1';
342         address <= adder_address_reg;
343
344         busy <= '1';
345         we_fu_mem_address <= '1';
346         fu_mem_address <= adder_address_reg;
347
348         next_state <= store_operation;
349
350       when store_operation =>
351         noe_early <= '1';
352         busy <= '0';
353         address <= adder_address_reg;
354         next_state <= idle;
355
356         -- Fetch unit requests access to the RAM
357         if (fetch_mem_req = '1') then
358           next_state <= fetch1;
359         end if;
360
361       when load1 =>
362         noe_early <= '0';
363         wr <= '0';
364         enable <= '1';
365         address <= adder_address_reg;
366
367         busy <= '1';
368         we_fu_mem_address <= '1'; -- Register the address
369         fu_mem_address <= adder_address_reg;
370         next_state <= load2;
371
372       when load2 =>
373         noe_early <= '0';
374         busy <= '1';
375         address <= adder_address_reg;
376         next_state <= load3;
377
378       when load3 =>
379         noe_early <= '0';
380         busy <= '0';
381         address <= adder_address_reg;
382         we_read_data_valid <= '1';
383         read_data_valid <= '1'; -- Next clk cycle register the read data
384         next_state <= idle;
385
386         -- Fetch unit requests access to the RAM
387         if (fetch_mem_req = '1') then
388           next_state <= fetch1;
389         end if;
390
391     end case;
392
393 end process;
394
395 -- SM updater
396 process(clk)
397 begin
398     if (rising_edge(clk)) then
399       if (rst = '1') then
400         state <= idle;
401       else
402         state <= next_state;
403       end if;
```

```
404   end if;
405 end process;
406
407 ------------------------------------------------------------------------
408 -- Optimized adder process to compute the address to access   --
409 ------------------------------------------------------------------------
410
411 i_reg_address : entity ulysse.reg generic map(register_width => memory_height) port map
412   (clk => clk, rst => rst, we => we_address, input => adder_address, output => adder_address_reg);
413
414 -- spl assert never( write_off_sp1 and load_off_sp1 ) report "Concurrent illegal axxs";
415 -- spl assert never( write_off_sp2 and load_off_sp2 ) report "Concurrent illegal axxs";
416 process(write_off_sp1, load_off_sp1, write_off_sp2, load_off_sp2, reg_sp1, reg_sp2, io_bus, ext_store, external_address)
417 begin
418 if(write_off_sp1 = '1' or load_off_sp1 = '1') then
419   --to_add <= reg_sp1;
420   adder_address <= reg_sp1 + io_bus(0)(data_memory_range);
421 elsif(write_off_sp2 = '1' or load_off_sp2 = '1') then
422   --to_add <= reg_sp2;
423   adder_address <= reg_sp2 + io_bus(0)(data_memory_range);
424 elsif(ext_store = '1') then
425   adder_address <= external_address;
426 else
427   adder_address <= io_bus(0)(data_memory_range);
428 end if;
429 end process;
430
431 end str;
```

Listing A.4: Memory unit code.

A.4 Assembler verbose output

```
_____________________________________________________

  Ulysse processor assembler - 2.31
  Pierre-Andre Mudry, EPFL 2004-2008
_____________________________________________________

File "output/test_segments_simple.remap" opened.
Starting assembly... success !

__Code Statistics___________________

Memory available : 262144 32-bits words
Data seg (static data only) memory usage : 0 %, 6 32-bits words
Code segment : 0.01 %, 27 instructions written
Stack reserved space : 12.5 % - 32768 32-bits words
Memory used : 12.51 % - 32801 words
Nop instructions : 0 %

__Labels____________________________

0 : the_end - Value : 19
1 : there - Value : 20
2 : here - Value : 16
3 : begin - Value : 0

__Data labels_______________________

0 : three - Value : 5
1 : ones - Value : 0

__Output files______________________

Memory (unified)  - "output/test_segments_simple" written
Data segment only - "output/test_segments_simple.mem" written
```

A.5 BNF grammar of the assembler

Note that this grammar is almost completely LL1[2], with the exception of labels and macro that have `identifier` as a prefix. This case is treated by a *look-ahead* of 2 to remove ambiguities.

```
<INCLUDE: "#include">
| <MOVE: "move">
| <DEF: "def">
| <NOP: "nop">
| <MACRO_START: "macro">
| <MACRO_END: "end">
| <PARA: "||">
| <IMM_SYMBOL: "#">
| <DATA_SEG: ".data">
| <CODE_SEG: ".code">
| <MEM_SIZE: ".mem_size">
| <STACK_SIZE: ".stack_size">

<OPERATOR: "+" | "−" | "&" | "|" | "<<" | ">>" | "^" | "∗" | "/">

<INTEGER_LITERAL: (["−"])? (<DECIMAL_LITERAL> (["l","L"])? | <HEX_LITERAL> (["l","L"])? | <
    BINARY_LITERAL> (["l","L"])?)>
| <#DECIMAL_LITERAL: ["0"−"9"] (["0"−"9"])∗>
| <#HEX_LITERAL: "0" ["x","X"] (["0"−"9","a"−"f","A"−"F"])+>
| <#BINARY_LITERAL: "0" ["b","B"] (["0"−"1"])+>

<IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)∗> | <#LETTER: ["$", "_","a"−"z","A"−"Z"]> | <#DIGIT:
    ["0"−"9"]>

<FILENAME: <LETTER> (<LETTER> | <DIGIT> | <SYMBOL>)∗> | <#SYMBOL: [".","−"]>
```

Listing A.5: Assembly grammar tokens.

```
Start          ::=  (Inclusion)∗ (ValDecl | MacroDef)∗ (Directive)∗ (((<DATA_SEG>(DataDecl)∗) | (<CODE_SEG
    >Program))∗) <EOF>
Inclusion      ::=  <INCLUDE> (("<"<FILENAME>">") | ("\""<FILENAME>"\"")) Start
MacroDef       ::=  "macro" <IDENTIFIER> "(" (<IDENTIFIER>(",")?)∗")" "macro" (";")?
ValDecl        ::=  <DEF> <IDENTIFIER> (Literal | Address) (";")∗
Directive      ::=  (<MEM_SIZE> | <STACK_SIZE>) Literal
DataDecl       ::=  Label (  (". dw" (<INTEGER_LITERAL> ",")∗ <INTEGER_LITERAL> (";")∗)+ | (".size" <
    INTEGER_LITERAL> (<INTEGER_LITERAL>)?))
Program        ::=  (Operation | Label)+
Operation      ::=  (FullMove | Nop | Macro)
Label          ::=  <IDENTIFIER> ":"
Macro          ::=  <IDENTIFIER> (TypedLiteral ("," )?)∗ ";"  (Operation)∗ <EOF>
FullMove       ::=  MoveOp ((<PARA> MoveOp ";") | (";" ))
MoveOp         ::=  "move" TypedLiteral "," TypedLiteral
Nop            ::=  ("nop"  ";")
Literal        ::=  (<INTEGER_LITERAL> | <IDENTIFIER>)
Address        ::=  "("  Literal  ","  Literal  ")"
Identifier     ::=  <IDENTIFIER>
TypedLiteral       ::=  ("#"  PrimaryExpression | PrimaryExpression)
PrimaryExpression ::= (Literal  (  <OPERATOR> PrimaryExpression)? | "(" PrimaryExpression ")" (<
    OPERATOR> PrimaryExpression)? )
```

Listing A.6: Assembly grammar non terminals.

[2]Which means that the reading of only one *token* is required to determine the current location in a production.

A.6 Processor simulation environment

The following text represents the output of an asserted simulation session within the MODELSIM™ simulator.

Figure A.1 depicts the complete simulation environment for the processor.

```
# ****************************************************
# *** Ulysse simulator 1.13, Pierre-André Mudry 2008
# ****************************************************
#
# * Clk since begin 1409 / Total clk 1409 - cycle 1
#    * Execute Move sp1, *1
# * Clk since begin 1413 / Total clk 1413 - cycle 2
#    * Execute Move sp2, *2
# * Clk since begin 1417 / Total clk 1417 - cycle 3
#    * Execute Move mem, *123
# * Clk since begin 1421 / Total clk 1421 - cycle 4
#    * Execute Move assert_line, *current_line(31)
# * Clk since begin 1425 / Total clk 1425 - cycle 5

[...]

# * Clk since begin 1713 / Total clk 1713 - cycle 69
#    * Execute Move assert_a_t, mem
# * Clk since begin 1717 / Total clk 1717 - cycle 70
#    * Execute Move assert_stop, *0
#
# ************************************************************
# *** Program end
# ************************************************************
# *** Clks cycles used since CPU restart or rst: 312
# *** 6.240000e+000 us @ 50MHz
# ************************************************************
```

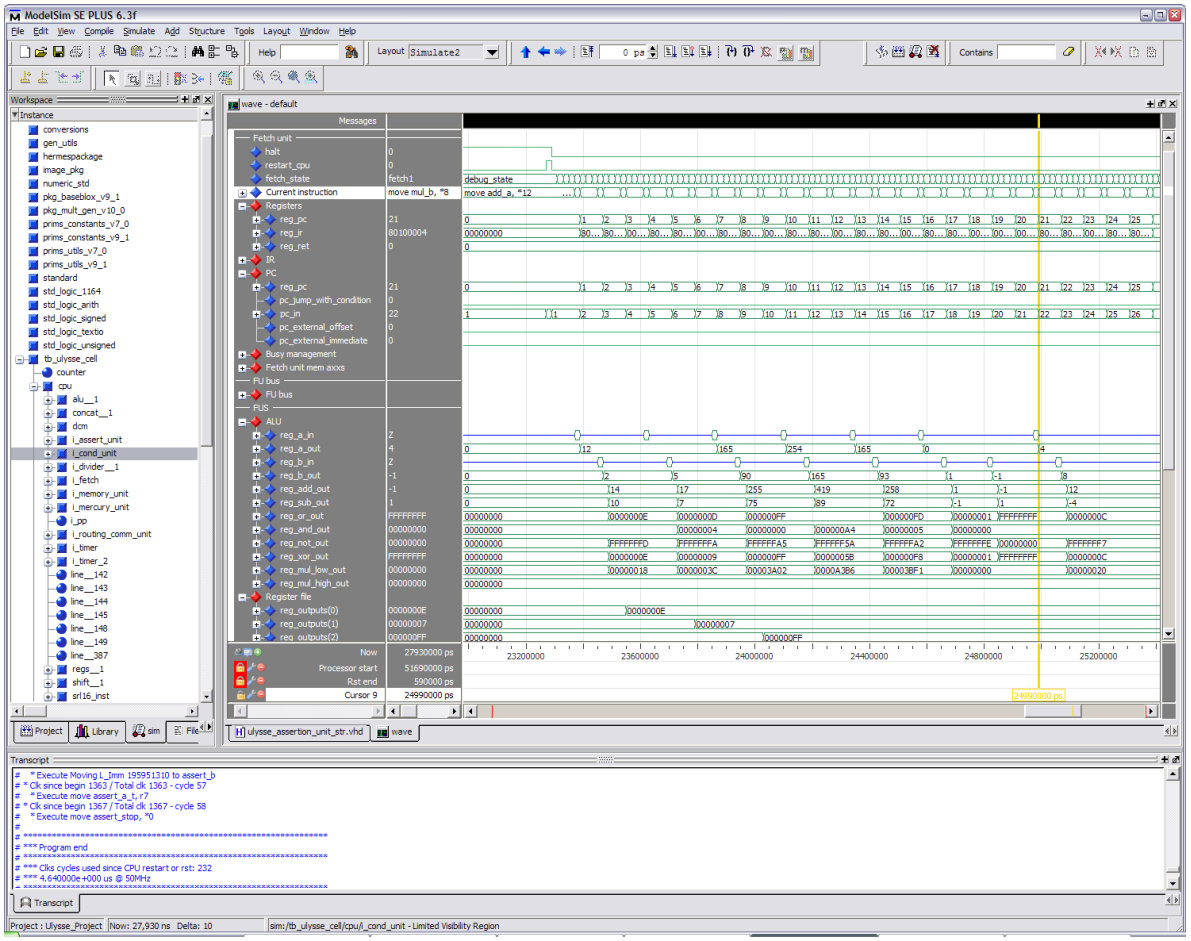


Figure A.1: Typical processor simulation and debugging session with MODELSIM™.

A.7 Benchmark code template for ULYSSE

```
#include "ulysse.h"

#define SIMULATION 1

// The function to be measured
int foo()
{
      [...]
}

void main()
{
 int i;
 int beginTime, endTime;

 get_timer1_us(beginTime);
    foo();
 get_timer1_us(endTime);

 #if SIMULATION_ONLY == 0
   clear_screen();
   display_bin(endTime-beginTime, 0);
 #else
   sim_assert_eq(block[0] == 87, 1);
   sim_exit();
 #endif

 // If we get to this point in simulation, something wrong happened
 sim_error();

 // Infinite loop
 while(1)
 {}
```

Listing A.7: Benchmark code template.

Appendix B

Software API appendix

B.1 Ulysse API

This API provides several C functions to access low-level capabilities of the ULYSSE processor without having to use assembly.

```
/****
 **** EPFL — IC — GRIJ
 **** Utility library of the Ulysse processor running on CONFETTI
 **** Rev 1.0
 **** Pierre—Andre Mudry, March 2008
 ****/

/***************************
 ** Low level macros and defines
 ***************************/

/* Reset Ulysse's timer unit. */
#define reset_timer1() \
    asm("move␣rst_timer_1,␣#(0);")

/* Reset Ulysse's _second_ timer unit. */
#define reset_timer2() \
    asm("move␣rst_timer_2,␣#(0);")

#define get_timer1_us(var) \
    asm("move␣%0,␣current_us_1;":"=r"(var));

#define get_timer2_us(var) \
    asm("move␣%0,␣current_us_2;":"=r"(var));

/* Display a pixel on the cell display
 * Address valid values is 8 bits
 *     Range 0..63      Red pixels
 *     Range 64..127    Green pixels
 *     Range 128..191 Blue pixels

 * Data valid values is 8 bits representing the brightness of each color
 * For example :
 * display(0, 255) displays a red pixel (max intensity) in the
 *  top left corner of the display
 * display(63, 127) displays a red pixel (half intensity) in the
 *  bottom right corner of the display
 * display(64, 255) displays a green pixel (max intensity) in the
 *  top left corner of the display
 */
#define display(address, data) \
    ({\
      asm("move␣display_address,␣%0;\nmove␣display_data,␣%1;"::"r"(address), "r"(data));\
     })
```

```
45
46 #define display_red(data) \
47     ({\
48       asm("move␣red,␣%0;"::"r"(data));\
49      })
50
51 #define display_green(data) \
52     ({\
53       asm("move␣green,␣%0;"::"r"(data));\
54      })
55
56 #define display_blue(data) \
57     ({\
58       asm("move␣blue,␣%0;"::"r"(data));\
59      })
60
61 // Returns the current cell x position (8 bit value) in the whole Confetti structure
62 #define get_x(x_pos) \
63     ({\
64       asm("move␣%0,␣x_pos;":"=r"(x_pos));\
65      })
66
67 // Returns the current cell y position (8 bit value) in the whole Confetti structure
68 #define get_y(y_pos) \
69     ({\
70       asm("move␣%0,␣y_pos;":"=r"(y_pos));\
71      })
72
73 /* Returns the value of the touch sensor (8 bit value) with the following meaning
74  * MSB ... LSB
75  * 0 0 S5 S4 S3 S2 S1 SR
76  * Bit 0 - SR : Sensor current state (debounced)
77  * Bit 1 - S1 : Means that the touch membrane has been activated during at least 1 sec
78  * Bit 2 - S2 : Means that the touch membrane has been activated during at least 2 sec
79  * Bit 3 - S3 : Means that the touch membrane has been activated during at least 3 sec
80  * Bit 4 - S4 : Means that the touch membrane has been activated during at least 4 sec
81  * Bit 5 - S5 : Means that the touch membrane has been activated during at least 5 sec
82  * Bit 6 - S6 : Means that the touch membrane has been activated during at least 6 sec
83  *
84  * Note that if the membrane is pressed during more than 5 seconds,
85  * the FPGA might be reconfigured from the routing layer
86  */
87 #define get_touch_sensor(touch_value) \
88     ({\
89       asm("move␣%0,␣touch_sensor;":"=r"(touch_value):);\
90     })
91
92 /*********************************************/
93 /* Simulation assertions                     */
94 /*********************************************/
95 // Report error with correct line of code in C
96 #define sim_assert_eq(a, b) \
97     ({\
98       asm("assert_eq_with_line␣%0,␣%1,␣%2;\
99 ␣␣␣␣␣//␣Report␣error␣with␣correct␣line␣of␣code␣in␣C"\
100       : \
101       :"r"(a), "r"(b), "i"(__LINE__));\
102     })
103
104 // Passes only if a <= b
105 void sim_assert_seq(int a, int b)
106 {
107   int result = a <= b;
108   sim_assert_eq(result, 1);
109 }
110
111 #define sim_error() {asm("error;");}
112 #define sim_exit() {asm("exit;");}
```

```
113
114 /*********************************************/
115 /* Mercury related                           */
116 /*********************************************/
117 #define get_mercury_status(stat) \
118     ({\
119       asm("move␣%0,␣mercury_status;":"=r"(stat));\
120      })
121
122 // Get the last packet size (which does not include the Mercury header)
123 #define get_mercury_packet_size(size) \
124     ({\
125       asm("move␣%0,␣mercury_packet_size;":"=r"(size));\
126      })
127
128 // Get the Confiture header of the last incoming packet
129 #define get_confiture_header(header) \
130     ({\
131       asm("move␣%0,␣confiture_header;":"=r"(header));\
132      })
133
134 // Get the DMA start address
135 #define get_dma_start_address(address) \
136     ({\
137       asm("move␣%0,␣mercury_dma_start_address;":"=r"(address));\
138      })
139
140 // Allows to receive packets and configuration
141 #define allow_reconfigure() {asm("move␣mercury_status,␣#(ACCEPT_DMA_MASK␣|␣
      ALLOW_RECONFIGURATION_MASK);");}
142
143 // Disable any incoming thing from Mercury
144 #define disable_reception() {asm("move␣mercury_status,␣0;")};
145
146 #define clear_nd_bit() {asm("move␣mercury_rst_new_data_flag,␣#1;");}
147
148 #define set_mercury_status(status) \
149     ({\
150       asm("move␣mercury_status,␣%0;"::"r"(data));\
151      })
152
153 // Send a packet throught the Mercury interface
154 #define m_send(send) \
155     ({\
156       asm("move␣mercury_send_data,␣%0;"::"r"(send));\
157      })
158
159 // Sets the address where the incoming packets from Mercury will be put
160 #define mDma_start_address(address) \
161     ({\
162       asm("move␣mercury_dma_start_address,␣%0;"::"r"(address));\
163      })
164
165 /**************************
166  ** Utility functions
167  **************************/
168 #define RED_PLANE 0
169 #define GREEN_PLANE 64
170 #define BLUE_PLANE 128
171
172 // Displays the to_display value as a binary number on the display
173 // plane_offset refers the three different color planes
174 void display_bin(int to_display, int plane_offset)
175 {
176   int i = 0;
177   int k = to_display;
178
179   // For each bit of the given parameter, display it
```

```
180   for(i = 0; i < 32; i++)
181   {
182     if(k & 0x1)
183         display(i+plane_offset, 255);
184     else
185         display(i+plane_offset, 0);
186
187     k >>= 1;
188   }
189 }
190 static unsigned long __next_rand = 1;
191
192 /* POSIX Implementation of rand */
193 /* RAND_MAX assumed to be 32767 */
194 int rand(void) {
195       __next_rand = __next_rand * 1103515245 + 12345;
196       return((unsigned)(__next_rand/65536) % 32768);
197  }
198
199 void set_seed(int seed)
200 {
201   __next_rand = seed;
202 }
203
204 void clear_screen()
205 {
206   for(int k = 0;k < 192; k++)
207   {
208     display(k, 0);
209   }
210 }
211
212 /**************************
213  ** Compiler functions
214  **************************/
215 void * memcpy(void * dst0, const void * src0, int n)
216 {
217   int *dst = (int *) dst0;
218   int *src = (int *) src0;
219   int  * save = dst0;
220
221   while(n)
222   {
223     *dst = *src;
224     src++;
225     dst++;
226     n--;
227   }
228
229   return save;
230 }
```

Listing B.1: Ulysse.h source file.

B.2 Messaging API

This is the complete API on how to use the messaging system of MERCURY. As explained in section 8.5, we used a bottom approach to build the messaging API: the MERCURY layer handles the low-levels calls whereas the CONFITURE layer provides functionalities such as broadcasting.

B.3 Mercury.h File Reference

Data Structures

- struct **packet**

  *Base **packet** type to encapsulate messages.*

Defines

- #define **PKT_HEADER_LENGTH** 1

  Packet header length.

- #define **PKT_TYPE_UNDEF** 0

  Undefined type (means any/none).

- #define **PKT_TYPE_ASYNC** 1

  One-way message (needs not be acked).

- #define **PKT_TYPE_SYNC** 2

  Synchronous message (needs to be ACKed).

- #define **PKT_TYPE_ACK** 3

  Acknowledge for the previous.

- #define **PKT_TYPE_SYNC_BCAST** 4

  Broadcast with ACKs.

- #define **PKT_TYPE_ASYNC_BCAST** 5

  Broadcast without ACKs.

- #define **PKT_TYPE_SYNC_BCAST1** 6

  First-level broadcast (send further along the same axis, send second-level broadcast into the other, perpendicular axis).

- #define **PKT_TYPE_SYNC_BCAST2** 7

  Second-level broadcast (send further along the same axis only.

- #define **PKT_TYPE_ASYNC_BCAST1** 8

  First-level broadcast (send further along the same axis, send second-level broadcast into the other, perpendicular axis).

- #define **PKT_TYPE_ASYNC_BCAST2** 9

  Second-level broadcast (send further along the same axis only.

- #define **PKT_TYPE_CODE_UPLOAD** 0xAB

  Reprogramming/code upload low-level message. DO NOT SEND THIS UNLESS YOU KNOW WHAT YOU ARE DOING.

- #define **read_mercury_status**(var) asm("move %0, mercury_status;":"=r"(var))

Typedefs

- typedef int **mercury_word_t**
- typedef unsigned int **mercury_size_t**
- typedef unsigned int **mercury_address_t**

Functions

- void **mercury_init** (**mercury_address_t** loc_addr, **mercury_address_t** inj_addr, int xmin, int xmax, int ymin, int ymax)

  Mercury init.

- int **mercury_send** (const **packet** *p)

  *Send a **packet** through the Mercury network.*

- int **mercury_send_possible** ()

  Check whether sending is possible with Mercury.

- int **mercury_receive** (**packet** *p, **mercury_size_t** buffer_size)

  *Get a **packet** from Mercury.*

- int **mercury_duplicate** (**mercury_address_t** target)

  Duplicate the code to another node.

- int **mercury_packet_available** ()

  *Check whether a **packet** is available from Mercury.*

- void **mercury_close** ()

  Close Mercury.

- int **mercury_valid_address** (**mercury_address_t** addr)

  Test whether the address is in the grid or not.

- **mercury_size_t mercury_exp_packet_size** (const **packet** *p)

  *Extracts the **packet** size from a raw Mercury **packet**.*

- void **print_packet** (const **packet** *p)

  Debugging function.

Variables

- int **local_addr**

  Address constants. They must be initialized correctly at runtime.

- **mercury_address_t north_addr**

  addresses for the NSEW neighbors.

- **mercury_address_t injection_addr**

  The address used by the injection point into the grid, to be considered valid !

- **mercury_address_t northeast_addr**

  Addresses for the diagonal neighbors.

- int **min_addr_x**

  Address limits. They have to be initialized correctly at runtime to delimit the node grid.

- int **grid_xdim**

  Updated dynamically to store the current Confetti limits.

B.3.1 Source code

```
#ifndef MERCURY_H
#define MERCURY_H

#include "util.h"

#ifdef TESTING
  #include <stdio.h>
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>
  #include <errno.h>
  #include <signal.h>
  #include <stdlib.h>
  #include <unistd.h>
#endif

/**********************/
/* PACKET DEFINITIONS */
/**********************/

/* primitive types */
typedef int mercury_word_t; /* meant to be 32 bits */
typedef unsigned int mercury_size_t; /* in mercury_word_t words */
typedef unsigned int mercury_address_t;

/* base packet type. might be resized to fit expanding requirements ...
   note that the first two fields must follow the conventions used by the underlying
   bus, ie. Mercury.

   Note that lengths and sizes are all in 32-bit words.

   Null values for addresses, types and sequence numbers
   mean "undefined" and are used as "any" values in conditional functions.
    */
typedef struct {
    /* fields are inverted in the real Hermes header.
       It seems our MOVE compiler stores bitfields the other way
       We would like them to be, on a 32-bit basis */
```

```
39
40      /* ought to be sender | length */
41      mercury_size_t length : 16; /* implies a maximal message payload of
42                                     64k-1 32-bit words: the header
43                                     takes 1 in this, so this length
44                                     is the real data length + 1.
45                                     This is a Mercury requirement.
46                                  */
47      mercury_address_t destination : 16; /* implies at most 256 destinations. */
48      /* ought to be: sender | type | seqnum, inverted to compensate for the compiler */
49      unsigned int seqnum : 8;
50      unsigned int type : 8; /* implies at most 256 packet types */
51      mercury_address_t sender : 16; /* implies at most 256 senders. */
52      mercury_word_t data[0]; /* pointer to the beginning of the payload, 32-bit words */
53  } __attribute__ ((__packed__)) packet;
54
55  /* Pack the structure as tight as possible with a __packed__ attribute.
56     Very important if the fields are not strictly byte-aligned */
57
58  /*  This length is defined as the length of the data that is not
59      taken into consideration in the "length" field of the packet struct.
60      Currently, this is 1 - the Mercury header itself only consists of the
61      32 first bits of the packet header we use. The rest of the struct is
62       specific and its length is taken into account in the length field.
63      */
64  #define PKT_HEADER_LENGTH 1
65
66
67  /*****************/
68  /* MESSAGE TYPES */
69  /*****************/
70
71  /* undefined type (means any/none) */
72  #define PKT_TYPE_UNDEF 0
73  /* one-way message (needs not be acked) */
74  #define PKT_TYPE_ASYNC 1
75  /* synchronous message (needs to be ACKed) */
76  #define PKT_TYPE_SYNC 2
77  /* acknowledge for the previous */
78  #define PKT_TYPE_ACK 3
79  /* Broadcast with ACKs */
80  #define PKT_TYPE_SYNC_BCAST 4
81  /* Broadcast without ACKs */
82  #define PKT_TYPE_ASYNC_BCAST 5
83  /* first -level broadcast (send further along the same axis,
84     send second-level broadcast into the other, perpendicular axis) */
85  #define PKT_TYPE_SYNC_BCAST1 6
86  /* second-level broadcast (send further along the same axis only */
87  #define PKT_TYPE_SYNC_BCAST2 7
88  /* first -level broadcast (send further along the same axis,
89     send second-level broadcast into the other, perpendicular axis) */
90  #define PKT_TYPE_ASYNC_BCAST1 8
91  /* second-level broadcast (send further along the same axis only */
92  #define PKT_TYPE_ASYNC_BCAST2 9
93
94  /* reprogramming/code upload low-level message.
95     DO NOT SEND THIS UNLESS YOU KNOW WHAT YOU ARE DOING. */
96  #define PKT_TYPE_CODE_UPLOAD 0xAB
97
98  /*   Initialize Mercury. This MUST be done before trying anything else.
99       Behavior is undefined otherwise.
100
101      */
102 void mercury_init(mercury_address_t loc_addr, mercury_address_t inj_addr,
103                   int xmin, int xmax, int ymin, int ymax);
104
105 /*  Send a packet through the Mercury network.
106
```

```
107     This function waits for the Mercury send buffer to have room, then sends
108     the packet proper, possibly blocking if Mercury itself blocks (full buffer).
109     A timeout mechanism prevents blocking forever. If this happens, ERROR_TIMEOUT
110     is returned, otherwise SUCCESS is returned when the transmission ends.
111     */
112 int mercury_send(const packet *p);
113
114 /* Check whether sending is possible with Mercury.
115     If the Mercury hardware layer reports this is possible, the function returns
116     TRUE. Otherwise, it returns FALSE.
117 */
118 int mercury_send_possible();
119
120 /* Get a packet from Mercury.
121
122     This function waits until a packet is available in the Mercury reception buffer.
123     Unless a timeout occurs, in which case it returns ERROR_TIMEOUT, it returns with
124     SUCCESS and the packet.This function can also return ERROR_BUFFER in the cases where the
125     entire packet is too large to fit in the given buffer size.
126     */
127 int mercury_receive(packet *p, mercury_size_t buffer_size);
128
129 /* Duplicate the code to another node.
130
131     This function sends a packet to the given node that reprograms it with
132     the sending node's code. For this to work, the CODE_BORDER constant has
133     to be declared at the beginning of the application code, before all
134     relevant includes (even mercury/confiture). All constant arrays and variables
135     declared BEFORE it will be sent, but those declared AFTER will be truncated
136     at send. */
137 int mercury_duplicate(mercury_address_t target);
138
139 /* Check whether a packet is available from Mercury.
140
141     If there is at least one packet in the receive buffer, return TRUE.
142     Otherwise, return FALSE.
143     */
144 int mercury_packet_available();
145
146 /* Close Mercury.
147
148     Of limited use in architectures that do not need to properly "quit"
149     an application using Mercury...
150     */
151 void mercury_close();
152
153 /* Test whether the address is in the grid or not. */
154 int mercury_valid_address(mercury_address_t addr);
155
156 mercury_size_t mercury_exp_packet_size(const packet *p);
157
158
159 /*******************/
160 /* DEBUG FUNCTIONS */
161 /*******************/
162 void print_packet(const packet *p);
163
164 #ifdef TESTING
165   /* Called in debug testing when the application gets a SIGINT */
166   void clean_exit();
167 #endif
168
169
170 /*****************************************/
171 /* CONSTANTS, GLOBAL VARIABLES, BUFFERS, ETC */
172 /*****************************************/
173
174 #ifndef TESTING
```

```
175   /* hardware-specific constants.
176        be careful to update them accordingly if they are changed! */
177   #define ACCEPT_DMA_MASK 0x00000001
178   #define ALLOW_RECONFIGURATION_MASK 0x00000002
179   #define WRITE_IN_PROGRESS_MASK 0x00000004
180   #define NEW_DATA_FLAG_MASK 0x00000008
181   #define CAN_SEND_DATA_MASK 0x00000010
182   #define RECEIVER_EMPTY_DATA_MASK 0x00000020
183
184   /* hardware defines */
185   //sample : asm("move %0, %1":"=r"(dest):"r"(src));
186   #define read_mercury_status(var) \
187        asm("move␣%0,␣mercury_status;":"=r"(var))
188   #define read_mercury_pkt_size(var) \
189        asm("move␣%0,␣mercury_packet_size;":"=r"(var))
190   #define read_confiture_header(var) \
191        asm("move␣%0,␣confiture_header;":"=r"(var))
192   #define write_mercury_status(var) \
193        asm("move␣mercury_status,␣%0;"::"r"(var))
194   #define write_mercury_status_val(val) \
195        asm("move␣mercury_status,␣%0;"::"i"(val))
196   #define write_mercury_data(var) \
197        asm("move␣mercury_send_data,␣%0;"::"r"(var))
198   #define write_dma_start_addr(var) \
199        asm("move␣mercury_dma_start_address,␣%0;"::"r"(var))
200   #define write_dma_end_addr(var) \
201        asm("move␣mercury_dma_end_address,␣%0;"::"r"(var))
202 #endif
203
204 #define ERROR_TIMEOUT 2
205 #define ERROR_BUFFER 3
206 #define ERROR_ADDRESS 4
207 #define ERROR_NOT_READY 5
208 #define ERROR_OVERFLOW 6
209
210 /* address constants. They must be initialized correctly at runtime */
211 int local_addr, local_addr_x, local_addr_y; /* address of self */
212 /* addresses for the NSEW neighbors */
213 mercury_address_t north_addr, south_addr, east_addr, west_addr;
214
215 /* the address used by the injection point into the grid,
216      to be considered valid ! */
217 mercury_address_t injection_addr;
218
219 //# ifdef MOORE_NEIGHBORHOOD
220 /* addresses for the diagonal neighbors */
221 mercury_address_t northeast_addr, southeast_addr, southwest_addr, northwest_addr;
222 //#endif
223
224 /* address limits. have to be initialized correctly at runtime
225    to delimit the node grid. */
226 int min_addr_x, max_addr_x, min_addr_y, max_addr_y;
227
228 /* grid dimensions, computed from the above. */
229 int grid_xdim, grid_ydim;
230
231 #ifndef TESTING
232   /* the MOVE GCC has no linker. */
233   #include "mercury.c"
234 #endif
235
236 #endif
```

Listing B.2: MERCURY API layer.

B.4 Confiture.h File Reference

Data Structures

- struct **cond_s**

  A condition structure used to match packets.

Defines

- #define **CONFITURE_MAX_PACKET_SIZE** 92

  Buffer size in 32-bit words.

- #define **CONFITURE_BUFFER_SIZE** 16

  Buffer size in packets.

Functions

- void **confiture_init** (**mercury_address_t** loc_addr, **mercury_address_t** inj_addr, int xmin, int xmax, int ymin, int ymax)

  Initialize the Confiture messaging layer.

- int **confiture_try_send** (**packet** *p)

  *Send a **packet** using Confiture.*

- int **confiture_receive** (**packet** *p, **mercury_size_t** buffer_size)

  *Receive a **packet** using Confiture.*

- int **confiture_check_receive** (**packet** *p, **mercury_size_t** buffer_size)

  *Receive a **packet**p. using Confiture.*

- int **confiture_wait_packet** (**packet** *p, **mercury_size_t** buffer_size, const **cond_s** *conditions, unsigned int ncond)

  *Wait for a **packet** of given type and sequence number, potentially buffering any **packet**p.structpacket that comes inbetween.*

- int **confiture_handle_packet_reception** (**packet** *p)

  Handle a packet's reception.

- int **confiture_propagate_broadcast** (**packet** *p)

  Propagate broadcast packets according to the algorithm we use.

- int **confiture_spread_code** ()

  Spread code further on the grid.

B.4.1 Source code

```
#ifndef CONFITURE_H
#define CONFITURE_H
#include "mercury.h"

/* buffer size in 32-bit words */
#ifndef CONFITURE_MAX_PACKET_SIZE
  #define CONFITURE_MAX_PACKET_SIZE 92
#endif

/* buffer size in packets */
#ifndef CONFITURE_BUFFER_SIZE
  #define CONFITURE_BUFFER_SIZE 16
#endif

/**************/
/* STRUCTURES */
/**************/
typedef struct {
    int sender;
    int type;
    int seqnum;
} cond_s;

/********************/
/* PUBLIC FUNCTIONS */
/********************/
void confiture_init (mercury_address_t loc_addr, mercury_address_t inj_addr,
                     int xmin, int xmax, int ymin, int ymax);

/* Send a packet using Confiture.

    This function is expected to block until: 1. the message could be
    transmitted correctly to the hardware layer, 2. any necessary answers
    have been obtained (ACK messages).

    Thus, if the function returns with value SUCCESS, one can be sure the
    message went through. When messages that use acknowledgements are used,
    one can even be sure the application received the message correctly.

    Timeouts are implemented to prevent blocking forever in failure cases.
    On such occasions, the function returns ERROR_TIMEOUT.

    See confiture_try_send() for a variant that does not wait for the hardware
    layer to be available.

    Be aware that the function can modify some fields
    (destination, sequence number) at will.
    */
int confiture_send(packet *p);

/* Send a packet using Confiture.

    This function does not wait until the hardware layer is ready, but returns
    FAILURE instead. If the hardware is available, it still waits until all
    proper answers have arrived, though, in which case it returns SUCCESS
    (or ERROR_TIMEOUT if a timeout occured).

    Be aware that the function can modify some fields
    (destination, sequence number) at will.
    */
int confiture_try_send(packet *p);

/* Receive a packet using Confiture.

    This function is expected to block until: 1. a message has been received,
    2. the message has been handled by Confiture. Only then the function returns
```

```
67      with SUCCESS and the message itself. Some timeouts might be useful to implement
68      in a way that the function never waits forever, returning ERROR_TIMEOUT instead.
69
70      This function can also return ERROR_BUFFER in the cases where the
71      entire packet is too large to fit in the given buffer size.
72
73      See confiture_check_receive() for a non-waiting variant of this function.
74      */
75  int confiture_receive(packet *p, mercury_size_t buffer_size);
76
77  /* Receive a packet using Confiture.
78
79      This function never blocks if no message is available, but returns FAILURE
80      instead. It still handles any message it receives.
81      */
82  int confiture_check_receive(packet *p, mercury_size_t buffer_size);
83
84  /* Wait for a packet of given type and sequence number, potentially buffering
85      any packet that comes inbetween.
86
87      Timeouts are used to prevent waiting forever. If this happens, the function
88      returns ERROR_TIMEOUT. Otherwise, it returns SUCCESS.
89
90      In the exceptional case the "overflow buffer" is too large, the function returns
91      ERROR_BUFFER.
92
93      If the function succeeds, a pointer to the satisfying packet is written in found_p.
94      */
95  int confiture_wait_packet(packet *p, mercury_size_t buffer_size,
96                            const cond_s *conditions, unsigned int ncond);
97
98
99  /**********************/
100 /* INTERNAL FUNCTIONS */
101 /**********************/
102
103 /* Handle a packet's reception, ie. send the necessary ACK messages
104     and propagate the message if needed.
105
106     If the ACK sending and possible broadcast are successful, the
107     function returns SUCCESS. Otherwise, FAILURE.
108     */
109 int confiture_handle_packet_reception(packet *p);
110
111 /* Propagate broadcast packets according to the algorithm we use.
112
113     If for any reason except reaching the end of the grid, a send fails,
114     then the function returns FAILURE. Otherwise, SUCCESS.
115     */
116 int confiture_propagate_broadcast(packet *p);
117
118 /* Spread code further on the grid.
119
120     To be used at startup if needed; does some kind of "broadcast" of
121     application code to speed up the initial programming process. If
122     put right after confiture_init(), this essentially means programming
123     a single cell is going to program the _whole_ grid.
124
125     Do NOT duplicate code that contains this in its startup sequence,
126     unless you want to reset the entire grid when doing so.
127     */
128 int confiture_spread_code();
129
130 /**********************/
131 /* AUXILIARY FUNCTIONS */
132 /**********************/
133 int confiture_match_packet(const packet *p, const cond_s *conditions, unsigned int ncond);
134
```

```
135 int confiture_buffer_empty();
136
137 int confiture_find_free_buffer();
138
139 #ifndef TESTING
140   /* the MOVE GCC has no linker. */
141   #include "confiture.c"
142 #endif
143
144 #endif
```

Listing B.3: CONFITURE API layer.

B.5 CAFCA code samples

```
// Spaceship motive
const int wss1[8][8] = {
        {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0},
        {0,0,1,1,1,0,0,0},    {0,0,1,1,0,0,1,0},    {0,0,0,0,1,1,1,0},
        {0,0,1,0,0,1,0,1},    {0,1,0,0,0,0,1,0}};

const int wss2[8][8] = {
        {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0},
        {0,0,0,0,1,0,0,0},    {1,1,0,1,1,0,0,0},    {0,0,0,1,0,0,0,0},
        {0,0,0,1,0,0,0,0},    {0,0,0,1,0,0,0,0}};

const int wss3[8][8] = {
        {0,1,0,0,0,0,1,0},    {0,0,1,0,0,1,0,1},    {0,0,0,0,1,1,1,0},
        {0,0,1,1,0,0,1,0},    {0,0,1,1,1,0,0,0},    {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0}};

const int wss4[8][8] = {
        {0,0,0,1,0,0,0,0},    {0,0,0,1,0,0,0,0},    {0,0,0,1,0,0,0,0},
        {1,1,0,1,1,0,0,0},    {0,0,0,0,1,0,0,0},    {0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0},    {0,0,0,0,0,0,0,0}};

const int CODE_BORDER = 0;

#include "../cafca/cafca.h" /* compile with -DMOORE_NEIGHBORHOOD */

#ifdef TESTING
  #include <stdio.h>
  #include <stdlib.h>
  #include <assert.h>
#endif

/* Grid dimensions */
#ifdef TESTING
  #define MIN_X 1
  #define MAX_X 3
  #define MIN_Y 1
  #define MAX_Y 3
#else
  #define MIN_X 1
  #define MAX_X 6
  #define MIN_Y 1
  #define MAX_Y 6
#endif

/* "injection" address for an interface outside the grid */
#define INJECTION_ADDR 0x0001

int state[8][8], new_state[8][8], north_state[8][8], east_state[8][8], west_state[8][8], south_state[8][8],
northeast_state[8][8], southeast_state[8][8], southwest_state[8][8], northwest_state[8][8];

/* Get a neighbor */
int neighbour8_wrapping(int i, int j) {
    if (i<0) {
        if(j < 0) return northwest_state[8+i][8+j];
        if(j > 7) return northeast_state[8+i][j-8];
        return north_state[8+i][j];
    }
    if (i>7) {
        if(j < 0) return southwest_state[i-8][8+j];
        if(j > 7) return southeast_state[i-8][j-8];
        return south_state[i-8][j];
    }
    if (j<0) return west_state[i][8+j];
    if (j>7) return east_state[i][j-8];
    return state[i][j];
```

```
}

void state_computation() {
    debug_msg("Computing state\n");
    debug_msg("neighbor states: %d %d %d %d %d %d %d %d\n",
            north_state, east_state, south_state, west_state,
            northeast_state, southeast_state, southwest_state, northwest_state);

    int weight;

    int i,j;
    for(i = 0 ; i < 8 ; i++) {
        for(j = 0 ; j < 8 ; j++) {
            weight = 0;

            weight += neighbour8_wrapping(i,j+1);
            weight += neighbour8_wrapping(i,j-1);

            weight += neighbour8_wrapping(i+1,j+1);
            weight += neighbour8_wrapping(i+1,j);
            weight += neighbour8_wrapping(i+1,j-1);

            weight += neighbour8_wrapping(i-1,j+1);
            weight += neighbour8_wrapping(i-1,j);
            weight += neighbour8_wrapping(i-1,j-1);

            if(state[i][j] == 0) {
                if(weight == 3) {
                    new_state[i][j] = 1;
                } else {
                    new_state[i][j] = 0;
                }
            } else {
                if(weight == 3 || weight == 2) {
                    new_state[i][j] = 1;
                } else {
                    new_state[i][j] = 0;
                }
            }
        }
    }

    memcpy(state, new_state, 64);
    debug_msg("New state = %d\n", state);
}

void display_state() {
  int i,j;

  #ifdef TESTING
    debug_msg("------------------------------\n");
    debug_msg("CURRENT STATE: ");
    printf("%d\n", state);
    debug_msg("------------------------------\n");
  #else

  /* if by any chance the membrane was pressed ...
     set the state to 1! */
  int sensor_value;
  get_touch_sensor(sensor_value);

  if(sensor_value & 0x1) {
     for(i = 0 ; i < 8 ; i++) {
         for(j = 0 ; j < 8 ; j++) {
             state[i][j] = 0;
         }
     }

```

```
134       // GLIDER
135       state [3][3] = 0; state [3][4] = 1; state [3][5] = 0; state [4][3] = 0;
136       state [4][4] = 0; state [4][5] = 1; state [5][3] = 1; state [5][4] = 1;
137       state [5][5] = 1;
138
139       // ACORN
140       /*const int acorn[8][8] = {
141             {0,0,0,0,0,0,0,0},
142             {0,0,0,0,0,0,0,0},
143             {0,0,1,0,0,0,0,0},
144             {0,0,0,0,1,0,0,0},
145             {0,1,1,0,0,1,1,1},
146             {0,0,0,0,0,0,0,0},
147             {0,0,0,0,0,0,0,0},
148             {0,0,0,0,0,0,0,0}
149       };*/
150
151       //memcpy(state, acorn, 64);
152   }
153
154   #ifndef DEBUG
155     for(i = 0 ; i < 8 ; i++) {
156       for(j = 0 ; j < 8 ; j++) {
157         display(RED_PLANE + i*8 + j, state[i][j]*255);
158         display(BLUE_PLANE + i*8 + j, state[i][j]*255);
159       }
160     }
161   #endif
162
163   #endif
164 }
165
166 void init() {
167   int i,j;
168   for(i = 0 ; i < 8 ; i++) {
169       for(j = 0 ; j < 8 ; j++) {
170           state[i][j] = 0;
171       }
172   }
173
174   // Copy the initial pattern
175   if(local_addr == 0x0302)
176       memcpy(state, wss1, 64);
177   else if(local_addr == 0x0402)
178       memcpy(state, wss2, 64);
179   else if(local_addr == 0x0301)
180       memcpy(state, wss3, 64);
181   else if(local_addr == 0x0401)
182       memcpy(state, wss4, 64);
183
184   clear_screen();
185
186   /* set CAFCA constants the way it ought to be */
187   app_state_size = 64;
188
189   app_state = (mercury_word_t*) state;
190   app_neighbor_states[0] = (mercury_word_t*) north_state;
191   app_neighbor_states[1] = (mercury_word_t*) east_state;
192   app_neighbor_states[2] = (mercury_word_t*) south_state;
193   app_neighbor_states[3] = (mercury_word_t*) west_state;
194   app_neighbor_states[4] = (mercury_word_t*) northeast_state;
195   app_neighbor_states[5] = (mercury_word_t*) southeast_state;
196   app_neighbor_states[6] = (mercury_word_t*) southwest_state;
197   app_neighbor_states[7] = (mercury_word_t*) northwest_state;
198
199   // Sets the display and state functions used by the CAFCA API
200   app_state_computation = &state_computation;
201   app_state_display = &display_state;
```

```
202 }
203
204 void main() {
205   cafca_init (0, 0x0101, &init, INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y);
206
207   // Self-replicate the code
208   if (local_addr == 0x0202)
209   {
210     confiture_spread();
211   }
212
213   cafca_main_loop();
214   cafca_close();
215 }
```

Listing B.4: Sample code for Game-of-life implemented with CAFCA.

```
1  // available values : jet512, hot512
2  #ifdef JET_PALETTE
3      #define PALETTE jet512
4      #include "../common/jet512.c"
5  #else
6      #define PALETTE hot512
7      #include "../common/hot512.c"
8  #endif
9
10 const int CODE_BORDER = 0;
11
12 #include "../cafca/cafca.h" /* compile with -DMOORE_NEIGHBORHOOD */
13 #ifdef TESTING
14   #include <stdio.h>
15   #include <stdlib.h>
16   #include <assert.h>
17 #endif
18
19 /* Grid dimensions */
20 #ifdef TESTING
21   #define MIN_X 1
22   #define MAX_X 3
23   #define MIN_Y 1
24   #define MAX_Y 3
25 #else
26   #define MIN_X 1
27   #define MAX_X 6
28   #define MIN_Y 1
29   #define MAX_Y 6
30 #endif
31
32 /* "injection" address for an interface outside the grid */
33 #define INJECTION_ADDR 0x0001
34
35 // Heat constants
36 #define DISSIPATION 1100
37
38 // Used to slow down the computation of the new states
39 // 0 is maximum speed
40 #define SLOWDOWN_FACTOR 2
41
42 int state [8][8], new_state[8][8], north_state [8][8], east_state [8][8], west_state [8][8], south_state [8][8],
43     northeast_state [8][8], southeast_state [8][8], southwest_state[8][8], northwest_state[8][8];
44
45 // Border condition where we simply copy the state of the border when pertinent
46 int get_copy(int i, int j) {
47
48     if (i<0) {
49       if (local_addr_y == MAX_Y) {
50           return state [0][ j ];
51       }
```

```
52          else
53          {
54            if(j < 0)
55            {
56                if(local_addr_x == MIN_X)
57                    return state[0][0];
58                else
59                    return northwest_state[8+i][8+j];
60            }
61            if(j > 7)
62            {
63                if(local_addr_x == MAX_X)
64                    return state[0][7];
65                else
66                    return northeast_state[8+i][j-8];
67            }
68            return north_state[8+i][j];
69          }
70      }
71
72      if (i>7){
73          if(local_addr_y == MIN_Y) {
74              return state[7][j];
75          }
76          else
77          {
78            if(j < 0)
79            {
80              if(local_addr_x == MIN_X)
81                return state[7][0];
82              else
83                return southwest_state[i-8][8+j];
84            }
85            if(j > 7)
86            {
87              if(local_addr_x == MAX_X)
88                return state[7][7];
89              else
90                return southeast_state[i-8][j-8];
91
92              // Random, comrade. Random.
93              set_seed(local_addr);
94            }
95
96            return south_state[i-8][j];
97          }
98      }
99
100     if (j<0) {
101       if(local_addr_x == MIN_X)
102         return state[i][0];
103       else
104         return west_state[i][8+j];
105     }
106
107     if (j>7) {
108       if(local_addr_x == MAX_X)
109         return state[i][7];
110       else
111         return east_state[i][j-8];
112     }
113     return state[i][j];
114 }
115
116 // Border condition with toroidal wrapping
117 int get64(int i, int j) {
118   if (i<0) {
119     if(j < 0) return northwest_state[8+i][8+j];
```

```
120     if(j > 7) return northeast_state[8+i][j-8];
121     return north_state[8+i][j];
122   }
123   if (i>7) {
124     if(j < 0) return southwest_state[i-8][8+j];
125     if(j > 7) return southeast_state[i-8][j-8];
126     return south_state[i-8][j];
127   }
128   if (j<0) return west_state[i][8+j];
129   if (j>7) return east_state[i][j-8];
130   return state[i][j];
131 }
132
133 void state_computation() {
134     int weight, i, j;
135
136     for(i = 0 ; i < 8 ; i++) {
137       for(j = 0 ; j < 8 ; j++) {
138         // If we are at the bottom of the Confetti system
139         // generate some nice random state that will make the whole thing
140         // look like fire
141         if ((i == 7) && local_addr_y == MIN_Y) {
142             if(j % 2 == 0) {
143                 int new = (rand() << 1) + (100 << 7);
144                 new_state[i][j] = min(new, (511<<7));
145             } else {
146                 new_state[i][j] = new_state[i][j-1];
147             }
148         }
149         else
150         {
151           weight = 0;
152
153           weight += get_copy(i,j+1);
154           weight += get_copy(i,j-1);
155
156           weight += get_copy(i+1,j+1);
157           weight += get_copy(i+1,j);
158           weight += get_copy(i+1,j-1);
159
160           weight += get_copy(i+2,j+1);
161           weight += get_copy(i+2,j);
162           weight += get_copy(i+2,j-1);
163
164           // u(t+1) = u(t) + du where du is the average of the neighbours - some dissipation
165           new_state[i][j] = ((weight >> 3) - state[i][j]) + state[i][j] - DISSIPATION;
166
167           // Be careful not to have negative temperatures ...
168           new_state[i][j] = max(0, new_state[i][j]);
169         }
170       }
171     }
172
173     if ((round_number & SLOWDOWN_FACTOR) == 0)
174         memcpy(state, new_state, 64);
175
176     debug_msg("New state = %d\n", state);
177 }
178
179 void display_state() {
180   #ifdef TESTING
181     debug_msg("------------------------------\n");
182     debug_msg("CURRENT STATE: ");
183     printf("%d\n", state);
184     debug_msg("------------------------------\n");
185   #else
186       /* if by any chance the membrane was pressed ...
187          set the state to 1! */
```

```
188         int sensor_value;
189         int i,j;
190         get_touch_sensor(sensor_value);
191
192         // make things a bit more random at the bottom (rand. button presses)
193         if ((sensor_value & 0x1) || ((local_addr_y == min_addr_y) && (rand() < 327))) {
194           state [3][3] = 0 xffff ;
195           state [4][4] = 0 xffff ;
196           state [3][4] = 0 xffff ;
197           state [4][3] = 0 xffff ;
198           state [2][3] = 0xfd00;
199           state [2][4] = 0xfd00;
200           state [3][2] = 0xfd00;
201           state [4][2] = 0xfd00;
202           state [3][5] = 0xfd00;
203           state [4][5] = 0xfd00;
204           state [5][3] = 0xfd00;
205           state [5][4] = 0xfd00;
206           state [2][2] = 0xf000;
207           state [2][5] = 0xf000;
208           state [5][2] = 0xf000;
209           state [5][5] = 0xf000;
210         }
211
212       #ifndef DEBUG
213         int color_bias;
214         if (local_addr_y > 3)
215           color_bias = 64;
216         else
217           color_bias = 0;
218
219         for(i = 0 ; i < 8 ; i++) {
220           for(j = 0 ; j < 8 ; j++) {
221             int clr = state[i][j] >> 7;
222
223             int pos = i*8 + j;
224
225             int red = max(((signed) PALETTE[clr][0]) - color_bias, 0);
226             int green = max(((signed) PALETTE[clr][1]) - color_bias, 0);
227             int blue = max(((signed) PALETTE[clr][2]) - color_bias, 0);
228             display(RED_PLANE + pos, red);
229             display(GREEN_PLANE + pos, green);
230             display(BLUE_PLANE + pos, blue);
231           }
232         }
233       #endif
234     #endif
235   }
236
237   void init () {
238     debug_msg("HELLO WORLD, INIT\n");
239
240     clear_screen();
241
242     // initialize the state as empty
243     int i,j;
244     for(i = 0 ; i < 8 ; i++) {
245         for(j = 0 ; j < 8 ; j++) {
246             state[i][j] = 0;
247         }
248     }
249
250     /* set CAFCA constants the way they ought to be */
251     app_state_size = 64;
252
253     app_state          = (mercury_word_t*) state;
254     app_neighbor_states[0] = (mercury_word_t*) north_state;
255     app_neighbor_states[1] = (mercury_word_t*) east_state;
```

```
256     app_neighbor_states[2] = (mercury_word_t*) south_state;
257     app_neighbor_states[3] = (mercury_word_t*) west_state;
258     app_neighbor_states[4] = (mercury_word_t*) northeast_state;
259     app_neighbor_states[5] = (mercury_word_t*) southeast_state;
260     app_neighbor_states[6] = (mercury_word_t*) southwest_state;
261     app_neighbor_states[7] = (mercury_word_t*) northwest_state;
262
263     // Sets the function that computes the next state
264     app_state_computation = &state_computation;
265
266     // And the display function
267     app_state_display = &display_state;
268
269     // Random, comrade. Random.
270     set_seed(local_addr);
271 }
272
273 void main() {
274     cafca_init (0, 0x0101, &init, INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y);
275
276     // Self-replication of the code
277     if (local_addr == 0x0202) {
278         confiture_spread();
279     }
280
281     cafca_main_loop();
282     cafca_close();
283 }
```

Listing B.5: Sample code for fire effect implemented with CAFCA.

B.6 Software architecture of the GUI

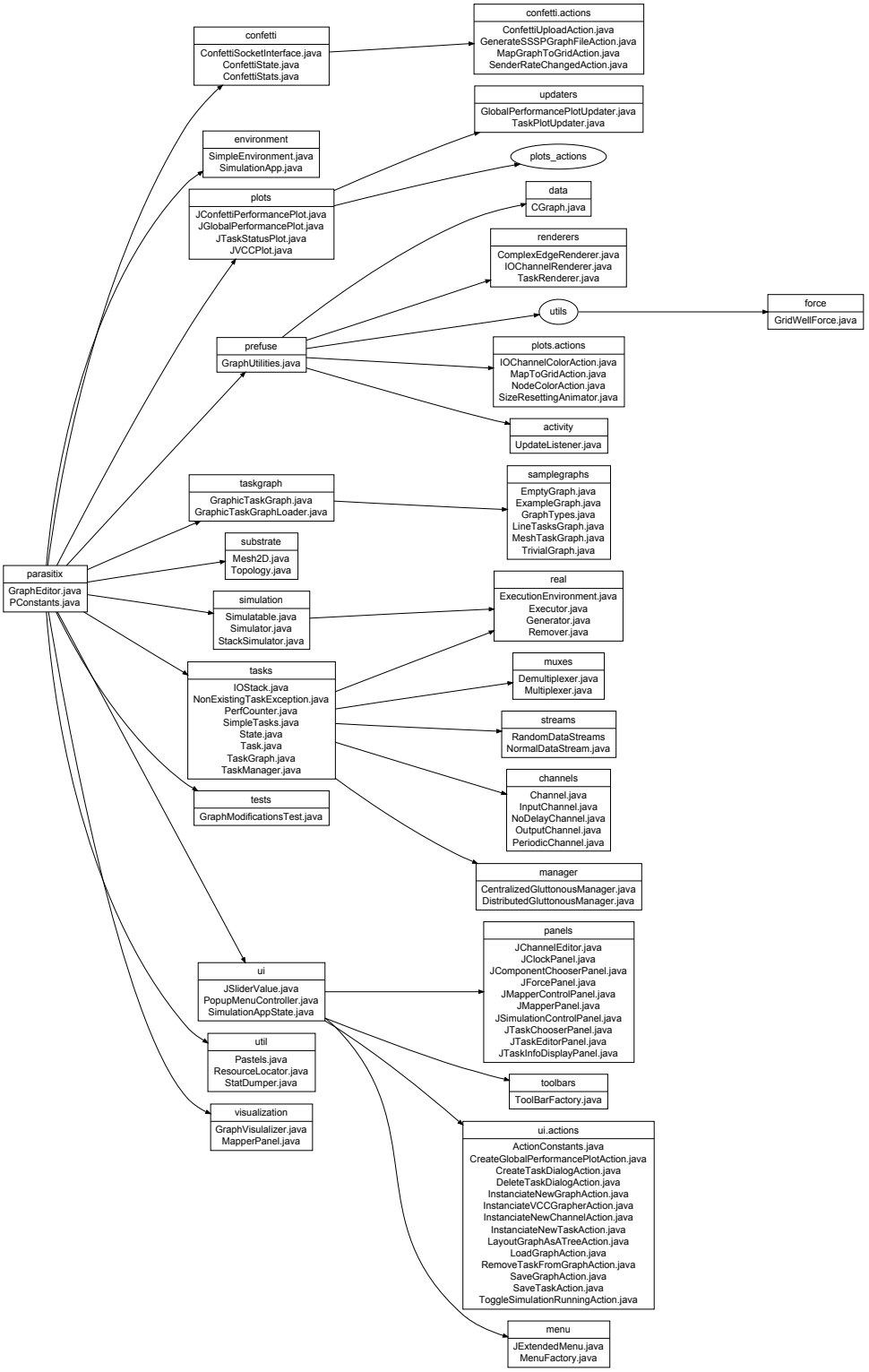


Figure B.1: Software architecture of the GUI.

B.7 CAFCA task code template

```
/*
** CAFCA Template
** For simulation only :
** Beware that every global variable declared here lies in the
** shared segment of the thread, which means that every task that implements
** this task has access to the SAME memory. Hence, every global data is
** shared among the instances of the thread.
**
** Pierre-Andre Mudry, EPFL, 2008
**/
#include "TASK_NAME.h"
#include "stdio.h"

// available values : jet512, hot512
#ifdef JET_PALETTE
    #define PALETTE jet512
    #include "../common/jet512.c"
#else
    #define PALETTE hot512
    #include "../common/hot512.c"
#endif

const int CODE_BORDER = 0;

#include "../cafca/cafca.h" /* compile with -DMOORE_NEIGHBORHOOD */

int state[8][8], new_state[8][8], north_state[8][8], east_state[8][8], west_state[8][8], south_state[8][8],
    northeast_state[8][8], southeast_state[8][8], southwest_state[8][8], northwest_state[8][8];

/*
* CAFCA task execution function
*/
void state_computation() {
    // TODO Insert your state computation code here
    int weight, i, j;

    for(i = 0 ; i < 8 ; i++) {
      for(j = 0 ; j < 8 ; j++) {
          new_state[i][j] = 0;
        }
    }

    memcpy(state, new_state, 64);
}

// Entry point for the task when executed in the simulation environment
#ifdef _SIM
  void init() {
    // Initialize the state as empty
    int i,j;
    for(i = 0 ; i < 8 ; i++) {
        for(j = 0 ; j < 8 ; j++) {
            state[i][j] = 0;
        }
    }

    /* set CAFCA constants the way they ought to be */
    app_state_size = 64;

    app_state          = (mercury_word_t*) state;
    app_neighbor_states[0] = (mercury_word_t*) north_state;
    app_neighbor_states[1] = (mercury_word_t*) east_state;
    app_neighbor_states[2] = (mercury_word_t*) south_state;
    app_neighbor_states[3] = (mercury_word_t*) west_state;
    app_neighbor_states[4] = (mercury_word_t*) northeast_state;
```

```
66      app_neighbor_states[5] = (mercury_word_t*) southeast_state;
67      app_neighbor_states[6] = (mercury_word_t*) southwest_state;
68      app_neighbor_states[7] = (mercury_word_t*) northwest_state;
69    }
70
71    void *task_init(void *taskParameters)
72    {
73      byte db = 0;
74      int status = C_NOCOMMAND;
75
76      stack instack, outstack;
77      sTask_init * initParams = (sTask_init *)taskParameters;
78
79      memset(instack.buffer, 0, BUFSIZE);
80      instack.level = 0;
81      instack.socket = 0;
82      instack.command = C_NOCOMMAND;
83      instack.port = initParams->inPort;
84
85      memset(outstack.buffer, 0, BUFSIZE);
86      outstack.level = 0;
87      outstack.socket = 0;
88      outstack.command = C_NOCOMMAND;
89      outstack.port = initParams->outPort;
90
91      init_sim_lib(&instack, &outstack);
92
93      printf("CAFCA Task created, name is %s, listening on port : %d and sending to port %d\n",
            initParams->taskname, instack.port, outstack.port);
94
95      while(1)
96      {
97        status = getStackStatus(&instack);
98
99        // If data present in input stack
100       if(status != C_NOCOMMAND)
101       {
102         printf("command received %x\n", status);
103       }
104       else if(!inputEmpty(&instack))
105       {
106           // Read data chunk
107           db = popInput(&instack);
108
109           // Wait until output stack can accept new data
110           while(outputFull()){};
111
112           // Push the result
113           pushOutput(state_computation(), &outstack);
114       }
115
116       // If simulated in a windows environment, sleep
117       #ifdef _WIN32
118       Sleep(1);
119       #endif
120     }
121
122     pthread_exit(NULL);
123   }
124 #endif
125
126 // Entry point for the task when executed in a standalone environment
127 #ifdef _PROFILING
128
129 void init() {
130   // Initialize the state as empty
131   int i,j;
132   for(i = 0 ; i < 8 ; i++) {
```

```
133         for(j = 0 ; j < 8 ; j++) {
134             state[i][j] = 0;
135         }
136     }
137
138     /* set CAFCA constants the way they ought to be */
139     app_state_size = 64;
140
141     app_state          = (mercury_word_t*) state;
142     app_neighbor_states[0] = (mercury_word_t*) north_state;
143     app_neighbor_states[1] = (mercury_word_t*) east_state;
144     app_neighbor_states[2] = (mercury_word_t*) south_state;
145     app_neighbor_states[3] = (mercury_word_t*) west_state;
146     app_neighbor_states[4] = (mercury_word_t*) northeast_state;
147     app_neighbor_states[5] = (mercury_word_t*) southeast_state;
148     app_neighbor_states[6] = (mercury_word_t*) southwest_state;
149     app_neighbor_states[7] = (mercury_word_t*) northwest_state;
150 }
151
152 void main()
153 {
154     // TODO Replace with your own profiling code
155     int i = 0;
156
157     init();
158
159     for(i = 0; i < 10000000; i++)
160     {
161       state_computation();
162     }
163 }
164 #endif
165
166 // Entry point for the task when executed in a standalone environment
167 #ifdef _REAL
168     void display_state() {
169
170       // TODO Insert the code to display the state here
171       int sensor_value;
172       int i,j;
173
174       /* if by any chance the membrane was pressed ...
175          set the state to 1! */
176       get_touch_sensor(sensor_value);
177
178       if(sensor_value & 0x1)
179         state [3][3] = 0xffff;
180         state [4][4] = 0xffff;
181         state [3][4] = 0xffff;
182         state [4][3] = 0xffff;
183       }
184
185       // Display the cell state using a palette
186       for(i = 0 ; i < 8 ; i++) {
187         for(j = 0 ; j < 8 ; j++) {
188           int clr = state[i][j] >> 7;
189           int pos = i*8 + j;
190
191           int red = max(((signed) PALETTE[clr][0]) - color_bias, 0);
192           int green = max(((signed) PALETTE[clr][1]) - color_bias, 0);
193           int blue = max(((signed) PALETTE[clr][2]) - color_bias, 0);
194           display(RED_PLANE + pos, red);
195           display(GREEN_PLANE + pos, green);
196           display(BLUE_PLANE + pos, blue);
197         }
198       }
199     }
200
```

```
201   void init () {
202     // TODO Change the init function to store the state size or use different neighborhood ...
203     clear_screen();
204
205     // Initialize the state as empty
206     int i,j;
207     for(i = 0 ; i < 8 ; i++) {
208         for(j = 0 ; j < 8 ; j++) {
209             state[i][j] = 0;
210         }
211     }
212
213     /* set CAFCA constants the way they ought to be */
214     app_state_size = 64;
215
216     app_state          = (mercury_word_t*) state;
217     app_neighbor_states[0] = (mercury_word_t*) north_state;
218     app_neighbor_states[1] = (mercury_word_t*) east_state;
219     app_neighbor_states[2] = (mercury_word_t*) south_state;
220     app_neighbor_states[3] = (mercury_word_t*) west_state;
221     app_neighbor_states[4] = (mercury_word_t*) northeast_state;
222     app_neighbor_states[5] = (mercury_word_t*) southeast_state;
223     app_neighbor_states[6] = (mercury_word_t*) southwest_state;
224     app_neighbor_states[7] = (mercury_word_t*) northwest_state;
225
226     // Sets the function that computes the next state
227     app_state_computation = &state_computation;
228
229     // And the display function
230     app_state_display = &display_state;
231
232     // Random, comrade. Random.
233     set_seed(local_addr);
234   }
235
236   void main() {
237     cafca_init (0, 0x0101, &init, INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y);
238
239     // Self-replication of the code (if required)
240     if (local_addr == 0x0202) {
241         confiture_spread();
242     }
243
244     // Call CAFCA main loop
245     cafca_main_loop();
246     cafca_close();
247   }
248 #endif
```

Listing B.6: Code template for a CAFCA task.

B.8 SSSP task code template

```
/**
** SSSP code TEMPLATE
**
** For simulation only :
** Beware that every global variable declared here lies in the
** shared segment of the thread, which means that every task that implements
** this task has access to the SAME memory. Hence, every global data is
** shared among the instances of the thread.
**
** Pierre-Andre Mudry, EPFL, 2008
**/

#include "TASK_NAME.h"
#include "stdio.h"

const int CODE_BORDER = 0;

#include "stream/stream.h"

// The task's execution function
int compute(int val)
{
  // TODO Replace with your own code
  int result = val + 2;
  return result;
}

// Entry point for the task when executed in the simulation environment
#ifdef _SIM
  void *task_init(void *taskParameters)
  {
    byte db = 0;
    int status = C_NOCOMMAND;

    stack instack, outstack;
    sTask_init * initParams = (sTask_init *)taskParameters;

    memset(instack.buffer, 0, BUFSIZE);
    instack.level = 0;
    instack.socket = 0;
    instack.command = C_NOCOMMAND;
    instack.port = initParams->inPort;

    memset(outstack.buffer, 0, BUFSIZE);
    outstack.level = 0;
    outstack.socket = 0;
    outstack.command = C_NOCOMMAND;
    outstack.port = initParams->outPort;

    init_sim_lib(&instack, &outstack);

    printf("Task created, name is %s, listening on port: %d and sending to port %d\n", initParams
        ->taskname, instack.port, outstack.port);

    while(1)
    {
      status = getStackStatus(&instack);

      // If data present in input stack
      if(status != C_NOCOMMAND)
      {
        printf("command received %x\n", status);
      }
      else if(!inputEmpty(&instack))
      {
```

```
65              // Read data chunk
66              db = popInput(&instack);
67
68              // Wait until output stack can accept new data
69              while(outputFull()) {};
70
71              // Push the result
72              pushOutput(compute(db), &outstack);
73          }
74
75          // If simulated in a windows environment, sleep
76          #ifdef _WIN32
77          Sleep(1);
78          #endif
79      }
80
81      pthread_exit(NULL);
82    }
83  #endif
84
85  // Entry point for the task when executed in a standalone environment
86  #ifdef _PROFILING
87    void main()
88    {
89      // TODO Replace with your own profiling code
90      int i = 0;
91
92      for(i = 0; i < 10000000; i++)
93      {
94        compute(i);
95      }
96    }
97    #endif
98
99    // Entry point for the task when executed in SSSP environment
100   #ifdef _REAL
101   void computation(job *j) {
102     // TODO Replace with your own code
103     my_job *aj = (my_job*) j->data; // Cast the job type.
104
105     // TODO Do you processing here
106     compute(YOUR_DATA);
107
108     // Display.
109     #ifdef DEBUG
110       display_bin(aj->state[DEBUG_POS], GREEN_PLANE+32);
111     #endif
112
113     // Correct the job type.
114     j->type = app_destination_gids[0];
115
116     // Send as-is. The sender is automatically corrected by
117     // the send() function.
118     stream_output(j);
119   }
120
121   void init () {
122     app_computation = &computation;
123     stream_status_array = status_array;
124   }
125
126   void main() {
127     stream_init(INJECTION_ADDR, MIN_X, MAX_X, MIN_Y, MAX_Y, &init);
128     stream_main_loop();
129   }
130 #endif
```

Listing B.7: Code template for a SSSP task.

List of Tables

List of Figures

List of Programs

List of Acronyms

ASIP	Application Specific Instruction set Processors
CISC	Complex Instruction Set Computer
CPI	Clocks Per Instruction
DAG	Directed Acyclic Graph
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
EMI	ElectroMagnetic Interference
EPIC	Explicitely-Parallel Instruction Computing
FDCT	Forward Discrete Cosine Transform
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GUI	Graphical User Interface
IDCT	Inverse Discrete Cosine Transform
IPC	Instructions Per Cycle
ILP	Instruction level parallelism
IO	Input Output
IR	Instruction Register
JNI	Java Native Interface
JTAG	Joint Test Action Group
LVDS	Low Voltage Differential Signalling
MPI	Message Passing Interface
MPSoC	Multi-processors Systems on Chip
NoC	Network on Chip
OOO	Out Of Order execution
PE	Processing element

PC Program Counter

PSL Property Specification Language

QoS Quality of Service

RF Register File

RISC Reduced Instruction Set Computer

SoC System-on-Chip

SIMD Single Instruction Multiple Data

SSA Static Single Assignment

TTA Transport Triggered Architecture

UART Universal Asynchronous Receiver Transmitter

VCC Variability Characterization Curve

VLC Variable Length Coding

VLIW Very Large Instruction Word

Pierre-André Mudry

PhD in Computer Science

Av. de Préfaully 38
1020 Renens, Switzerland
✉ *pierre-andre.mudry@a3.epfl.ch*
+41-79-507.89.36
☎ *+41-21-634.81.89*
31 years old
Swiss citizen

Education

2004–2009 **PhD in Computer Science**, *École Polytechnique Fédérale*, Lausanne.
A Hardware-Software Codesign Framework for Cellular Computing

1998–2004 **Master in Computer Science**, *École Polytechnique Fédérale*, Lausanne.

1993-1998 **Maturité cantonale**, *Lycée-collège de St-Guérin*, Sion.
Orientation socio-économique (type E)

Languages

French	**Native speaker**	
German	**Basic**	*Basic reading and writing skills. Steadily improving in speaking.*
English	**Advanced**	*Fluent in writing and speaking. Daily use.*
Italian	**Basic**	*Simple written and oral understanding, limited speaking.*

Skills by domain

Hardware development

Architectures	Development of DSP and MOVE processors, cellular architectures
Processors	MOVE, ARM7, 8051, PIC, DSP, Microblaze
Protocols	USB, I^2C, SPI, PCI, 1-Wire
PCB	Schematic and routing (*Altium Designer* and *Orcad*). Fabrication
Quality insurance	Symbolic debugging. Code regressions and multi-level code assertion

Programming and EDA

Systems	Linux, Windows	Documents	Matlab, LaTeX, MS Office
Programming	Java, C, C++, assembly	GUI	Visual Basic, C++, Swing
EDA	**VHDL**, Modelsim, Xilinx EDK and ISE, Synplify	Web	HTML, CSS

Work Experience

Consulting

2006–2009 **Electronics consultant**, *Jilli Consulting*, Lausanne.
Schematic design, complex PCB routing.

Teaching

2007 **Digital design course**, *Substitute lecturer*, HEIG-VD, Yverdon.
Course lecturing and lab supervision.

Assistantships

2007 **Advanced digital design**, *Head assistant*.
Lab session support, exam preparation and supervision.

2005–2006 **Logic systems**, *Head assistant*.
Lab session support, exam correction.

2005–2006 **Microprogrammed systems**, *Head assistant*.
Lab preparation and support.

2004–2005 **C++ language**, *Head assistant*.
Lecturing and lab supervision.

2002–2004 **Miscellanous**, *Student assistant*.
Computer Architecture I & II, Elementary Logics, Theory of Computer Science.

Internships

Sept. and Oct. 2002 **Research assistant**, *Logic systems laboratory – EPFL*, Lausanne.
Development of a GUI for evolutionary algorithms.

Sept. and Oct. 2003 **VHDL programmer**, *CSEM*, Neuchâtel.
Realization and test of the Macgic DSP ALU.

Publications

Conference Papers

2008 Pierre-André Mudry, Julien Ruffin, Michel Ganguin & Gianluca Tempesti. *A hardware-software design framework for distributed cellular computing*. In Proceedings of the 8th International Conference on Evolvable Systems (ICES'08), pages 71–82, Prague, September, 2008.

Pierre-André Mudry, Sarah Degallier & Aude Billard. *On the influence of symbols and myths in the responsibility ascription problem in roboethics - A roboticist's perspective*. In Proceedings of the 17th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN'08), pages 563-568, Münich, August, 2008.

Andrej Gams & Pierre-André Mudry. *Gaming controllers for research robots: controlling a humanoid robot using a WIIMOTE*. In Proceedings of the 17th International Electrotechnical and Computer Science Conference (ERK'08), Portorož, September, 2008. To appear.

2007 Pierre-André Mudry, Fabien Vannel, Gianluca Tempesti & Daniel Mange. *CONFETTI : A reconfigurable hardware platform for prototyping cellular architectures*. In Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'07), pages 1–8, Long Island, March, 2007.

2006 Pierre-André Mudry, Guillaume Zufferey & Gianluca Tempesti. *A dynamically constrained genetic algorithm for hardware-software partitioning*. In Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO'06), pages 769–776, Seattle, July, 2006.

Gianluca Tempesti, Daniel Mange, Pierre-André Mudry, Joël Rossier & André Stauffer. *Self-replication for reliability: bio-inspired hardware and the embryonics project*. In Proceedings of the 3rd Conference on Computing frontiers, pages 199–206, Ischia, 2006.

Gianluca Tempesti, Pierre-André Mudry & Guillaume Zufferey. *Hardware/software coevolution of genome programs and cellular processors*. In Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06), pages 129–136, Istanbul, June, 2006.

Pierre-André Mudry, Guillaume Zufferey & Gianluca Tempesti. *A hybrid genetic algorithm for constrained hardware-software partitioning*. In Proceedings of the 2006 IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pages 3–8, Prague, April, 2006.

Joël Rossier, Yann Thoma, Pierre-André Mudry & Gianluca Tempesti. *MOVE processors that self-replicate and differentiate*. In Proceedings of the 2nd International Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT'06), pages 160–175, Osaka, January, 2006.

2005 Gianluca Tempesti, Pierre-André Mudry & Ralph Hoffmann. *A MOVE processor for bio-inspired systems*. In Proceedings of the NASA/DoD Conference on Evolvable Hardware (EH'2005), pages 262–271, Washington, June, 2005.

Journal papers

2007 Gianluca Tempesti, Daniel Mange, Pierre-André Mudry, Joël Rossier & André Stauffer. *Self-replicating hardware for reliability: The embryonics project*. In ACM Journal on Emerging Technologies in Computing Systems (JETC), volume 3, number 2, 2007.